VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Spektrální klastrování a jeho paralelizace pomocí technologie CUDA

Spectral Clustering and its Parallelization using CUDA Technology

2014

Michal Čerbák

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Michal Čerbák**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Spektrální klastrování a jeho paralelizace pomocí technologie CUDA
Spectral Clustering and its Parallelization using CUDA Technology

Zásady pro vypracování:

V posledních letech se spektrální klastrování stalo jednou s nejpoužívanějších metod. Využívá jednoduché prostředky lineární algebry. Úkolem této práce je s využitím technologie CUDA tuto metodu paralelizovat.

1. Student nastuduje spektrální klástrování a naimplementuje ho.
2. Experimentálně prověří svou implementaci na segmentaci obrazu.
3. Svou implementaci nakonec zparalelizuje pomocí technologie CUDA.

Seznam doporučené odborné literatury:

[1] Andrew Y. Ng and Michael I. Jordan and Yair Weiss, "On Spectral Clustering: Analysis and an algorithm", ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS, 2001, p. 849-856, MIT Press
[2] NVIDIA CUDA C Programming Guide, July 2013, http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Karel Mozdřeň**

Datum zadání: 01.09.2013

Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.*

V Ostravě 5. mája 2014 ..... Michal Cirbad

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 5. mája 2014 ..... Michal Cirbad

Rád by som na tomto mieste poďakoval pánovi Ing. Karlovi Mozdřeňovi, ktorý mi umožnil pochopiť problematiku a svojimi cennými radami a skúsenosťami mi pomohol vytvoriť túto prácu, ktorá by bez neho vzniknúť nemohla. Ďalej by som chcel poďakovať svojím rodičom, ktorí ma pri štúdiu podporovali. A v neposlednej rade patrí poďakovanie mojej priateľke, ktorá trpela moje nálady v dňoch dokončovania práce.

Abstrakt

Táto práca sa zaoberá problémom spektrálneho klastrovania a jeho paralelizácie pomocou technológie CUDA. V práci je popísaná a rozobraná problematika segmentácie obrazu, ktorá je riešená spektrálnym klastrovaním. Práca osvetľuje problematiku klastrovania a popisuje princípy a postupy riešenia teoreticky. Detailne je rozobratý praktický návrh riešenia. Navrhnuté je sekvenčné a paralelné riešenie, použité metódy a vylepšenia, ktoré boli vyvinuté na zlepšenie časovej a pamäťovej náročnosti. Práca sa zaoberá aj porovnaním spracovaných riešení a vysvetľuje ich rozdiely.

Klíčová slova: spektrálne klastrovanie, paralelizácia, CUDA, segmentácia obrazu

Abstract

This paper deals with spectral clustering problem and its parallelization using CUDA technology. Paper also describes image segmentation problem, which is solved using spectral clustering. Clustering problem is outlined as well as theoretical principles and processes of final solution. Practical solution is described in detail. Sequential and parallel solution are proposed as well as used methods and tweaks designed to decrease time and memory requirements. Paper also focuses on comparing processed solutions and describes their differences.

Keywords: spectral clustering, parallelization, CUDA, image segmentation

Seznam použitých zkratek a symbolů

CUDA	–	Computing Unified Device Architecture
OpenCL	–	Open Computing Language
GPU	–	Graphics Processing Unit
GPGPU	–	General Purpose computing on Graphics Processing Units
DSP	–	Digital signal processor
ARM	–	Acorn RISC Machine
RISC	–	Reduced Instruction Set Computer
GDDR	–	Graphics Double Data Rate
SIMD	–	Single Instruction Multiple Data
CUBLAS	–	CUDA Basic Linear Algebra Subroutines
CULA	–	CUDA Linear Algebra

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

V dnešnej dobe je čoraz viac využívaná automatizácia všetkých procesov, ktoré sa automatizovať dajú. K automatizácii veľmi prispieva odvetvie spracovania obrazu nazývané počítačové videnie. Jednou z metód používaných v počítačovom videní je segmentácia obrazu. Ide o postupy pri ktorých je obraz spracovaný tak, aby počítač dokázal oblasti v obraze od seba odlíšiť na základe stanovených kritérií a metrík. Existuje množstvo segmentačných metód. Táto práca sa zaoberá metódou spektrálneho klastrovania.

Spektrálne klastrovanie je jednou z metód využívaných na segmentovanie dát. Táto metóda je veľmi náročná na strojový čas a pamäť. V tejto práci som sa snažil navrhnúť riešenie spektrálneho klastrovania pomocou technológie NVIDIA CUDA a využiť tak výkon grafickej karty, ktorý je v porovnaní s procesorom väčšinu času nevyužitý. Dnešné grafické akcelerátory poskytujú stovky výpočetných jadier za zlomok ceny ekvivalentne výkonného procesoru. Preto technológia CUDA dokáže ušetriť náklady na hardware ale aj na energiu, vďaka zrýchleniu algoritmov oproti bežnému procesoru.

V nasledujúcej kapitole sú popísané paralelné výpočty, ich využitie, problémy ktoré vznikajú pri paralelizácii a prístupy k paralelnému programovaniu. Popísané sú aj dostupné paralelné riešenia. Tretia kapitola je venovaná histórii technológie CUDA a technologickému vývoju, ktorý GPGPU[1] výpočtom predchádzal. Štvrtá kapitola rozoberie technologické pozadie CUDA programovania. Preberie základné princípy, ktoré sa v CUDE používajú. Predstavená v nej bude hierarchia vlákien a blokov ktoré sú základom paralelizácie. Popísané budú aj druhy pamäte, ktorými CUDA zariadenie disponuje, rovnako ako aj overené postupy a chyby, ktoré sa pri práci s pamäťou vyskytujú. V piatej kapitole bude popísaný teoretický základ segmentácie obrazu, ako aj jeho využitia. Šiesta kapitola vysvetlí a popíše jednotlivé kroky spektrálneho klastrovania, od vytvorenia matíc Affinity, Diagonálnej a Laplaceovej až po klastrovanie. Všetky postupy sú detailne a jednoducho vysvetlené. V siedmej kapitole bude predstavené praktické riešenie problému spektrálneho klastrovania. Budú popísané jednotlivé postupy, ktoré boli pri implementácii použité. Detailnejšie bude popísaná metóda power iteration, ktorá je využitá na výpočet vlastných čísel a vektorov. Predstavené bude sekvenčné a paralelné riešenie technológiou CUDA. V ôsmej kapitole bude overená správnosť návrhu riešení. Správnosť bude posudzovaná z hľadiska časovej a pamäťovej náročnosti, na základe ktorých budú sekvenčné a paralelné riešenie porovnávané. Objavia sa tu aj výsledné segmentácie obrazu. Deviata kapitola patrí diskusii, v ktorej budú prebraté moje subjektívne názory na tematiku a na výsledky mojej práce. Poslednou desiatou kapitolou je záver, kde bude zhrnutý prínos tejto práce.

[1] Výpočty na GPU, ktoré priamo nesúvisia s vykresľovaním obrazu

2 Paralelné výpočty

Paralelné výpočty sú v informatike označenia pre výpočty, ktoré môžu byť riešené súbežne. Paralelizácia je využívaná na zvýšenie výpočetného výkonu v situácii, keď sa nedá využiť rýchlejší počítač, na zníženie výpočetnej zložitosti použitého algoritmu, alebo na zníženie nákladov spojených so strojovým časom potrebným na riešenie daného problému. Na zrýchlenie sa využívajú buď viacprocesorové systémy alebo počítačové klastre[2]. Významným pojmom v oblasti paralelizácie je škálovateľnosť. Ide o veľmi dôležitú vlastnosť paralelného systému alebo procesu reagovať na zmenu, v zmysle adekvátneho zvýšenia výpočtového výkonu pri zvýšení hardwareových prostriedkov.

Paralelné výpočty môžu byť delené podľa viacerých kritérií. Podľa foriem paralelizmu na funkčný paralelizmus, ktorý je vo viacjadrovom systéme dosiahnutý, keď každý procesor vykonáva rozličné operácie na rovnakých alebo rôznych dátach. Oproti tomuto prístupu stojí dátový paralelizmus, ktorého procesory vykonávajú rovnaké operácie nad rovnakými alebo rôznymi dátami.

Podľa úrovne paralelizmu: jeden počítač spracovávajúci dáta viacerými procesormi prípadne jadrami naraz, alebo klastre, MPP a gridy pozostávajúce z množstva počítačov pracujúcich na rovnakej úlohe. Špecializované paralelné počítačové architektúry sú často využívané vedľa tradičných procesorov iba na urýchlenie konkrétnych častí výpočtov, na ktoré sa paralelizmus hodí.

Paralelné programy sú implementačne náročnejšie ako bežné sekvenčné algoritmy, pretože súbežnosť vytvára množstvo potencionálnych úskalí. Najčastejšími sú komunikácia, synchronizácia a konkurencia medzi jednotlivými úlohami, čo býva najväčšou prekážkou pri vývoji paralelného programu a negatívne sa odráža na jeho výkone [1].

Sľubný prístup na dosiahnutie vysokého výkonu vo výpočtoch, je využívanie systémov s extrémnym stupňom paralelizácie. Takýmito systémami sú GPU - ich masívny paralelizmus využívajú technológie CUDA a OpenCL, ktoré poskytujú multivláknové GPGPU programovanie. Ďalšími takýmito systémami sú viacjadrové DSP a ARM systémy, ale aj heterogénne výpočtové systémy. Zo sofwareového uhlu pohľadu je nutné spomenúť najpoužívanejší prístup k distribuovaným paralelným výpočtom na viacjadrových procesoroch - MPI (message passing interface), ktoré špecifikuje komunikáciu medzi oddelenými procesormi. MPI bolo navrhnuté ako na masívne paralelné stroje, tak aj na klastre z bežných počítačov. Odlišný prístup reprezentuje OpenMP s možnosťou paralelizovania programu vrámci jedného viacjadrového/procesorového stroja s použitím zdieľaného pamäťového priestoru. OpenMP je multiplatformové API pre paralelné programovanie.

[2] Zoskupenie počítačov, ktoré spolu úzko spolupracujú, takže sa navonok môžu tvárit' ako jeden počítačový systém. Využívané na zvýšenie výpočetného výkonu a spoľahlivosti.

3 História CUDY

Prvé GPU boli navrhnuté len ako grafické akcelerátory, podporovali len pevne zadané funkcie na transformácie vrcholov a fragmentov(na rozdiel od dnešných plne programovateľných shader procesorov). Od 1990 roku sa ale hardware začal stávať viac programovateľný, čoho výsledkom bola prvá NVIDIA GPU v 1999. Menej ako rok potom čo NVIDIA vytvorila pojem GPU, neboli herní vývojári jediní, ktorých práca na tejto technológii pohltila: Vedci začali objavovať obrovský výkon pri operáciách s plávajúcou desatinnou čiarkou. Začali vznikať základy GPGPU výpočtov.

Skutočné GPGPU ale bolo od tohto bodu ešte veľmi ďaleko, aj pre tých ktorí poznali shaderovacie jazyky ako OpenGL. Vývojári museli mapovať ich matematické výpočty na problémy, ktoré mohli byť reprezentované trojuholníkmi alebo polygónmi. GPGPU bolo prakticky tabu pre tých, ktorí si nepamätali posledné grafické API, až do okamihu keď vedci zo Stanfordskej univerzity začali vnímať GPU ako streaming processor.

V roku 2003 predstavil tím vedcov, vedený Ian Buckom, Brook, prvý programovací model rozširujúci jazyk C využívajúci paralelizmus. Používal koncepty ako streamy, kernely, redukciu. Brook ukázal GPU ako procesor určený na iné ako grafické problémy (general purpose processor) v high level jazyku[3] Čo ale bolo najdôležitejšie: Programy v Brooku neboli len jednoduchšie na písanie ako ručne upravený GPU kód ale boli 7 krát rýchlejšie ako podobný existujúci kód.

NVIDIA vedela, že s rýchlym hardwareom sú spojené intuitívny software a hardwareové nástroje, preto zamestnala Iana Bucka aby začal vyvíjať riešenie ako spustiť kód C na GPU. V roku 2006 NVIDIA spojila hardware a software a predstavila technológiu CUDA, prvú architektúru pre výpočty GPGPU. [3].

[3]Programovací jazyk so silnou abstrakciou od hardwaru počítača. V porovnaní s low level jazykom, môže byť jednoduchší na používanie. Môže zautomatizovať alebo úplne skryť časti programov, napríklad: réžia pamäte, čo môže celý proces vývoja programu spraviť jednoduchším a viac pochopiteľným.

4 Architektúra CUDA

Ako bolo spomínané v kapitole 3 technológia CUDA bola vyvinutá spoločnosťou NVIDIA a zakladá svoju silu na obrovskom paralelnom výkone ich grafických kariet, ktoré pri svojej relatívne nízkej cene, poskytujú mnohonásobne vyšší výkon v porovnaní s cenovo podobnými procesormi.

CUDA ukazuje svoju silu predovšetkým v algoritmoch, ktoré spracovávajú veľké množstvo dát. Ideálne je využitie ak sa nad dátami vykonávajú SIMD[4] operácie, pretože vtedy dokáže CUDA plne využiť potenciál garfickej karty. Takéto nasadenie CUDY je v širokom spektre aplikácií od prehrávania 4K videa v reálnom čase, jeho dekódovania cez komprimačné algoritmy až po zložité algoritmy rozpoznávania obrazu a iné.

Paralelný program využívajúci CUDU môže byť implementovaný v najznámejších high level jazykoch C/C++, Python, .NET, Matlab a iných. Princíp programovania v CUDE je založený na rozdeľovaní problémov na menšie časti, ktoré budú riešené samostatne. Tieto čiastkové problémy musia byť riešiteľné nezávisle na zvyšku problému pretože môžu byť spracované paralelne blokmi (záleží na konfigurácii a hardware). V blokoch sú tieto zjednodušené časti opäť rozdelené ale medzi vlákna, ktoré ich riešia paralelne. Takéto usporiadanie dovoľuje vláknam spolupracovať vrámci bloku ale bloky musia byť na sebe nezávislé, pretože GPU ich môže riešiť paralelne, ale aj postupne a v rôznom poradí. Tento model umožňuje škálovateľnosť programu, to znamená že program nemusí byť písaný pre konkrétny hardware a pri spustení na výkonnejšom GPU bude aj výkon aplikácie vyšší. To umožňuje pokryť celú škálu GPU, od mobilných a desktopových cez výkonné hráčske karty až po profesionálne akcelerátory Tesla a Quadro, ktoré sú stavané primárne na GPGPU.

4.1 Kernely

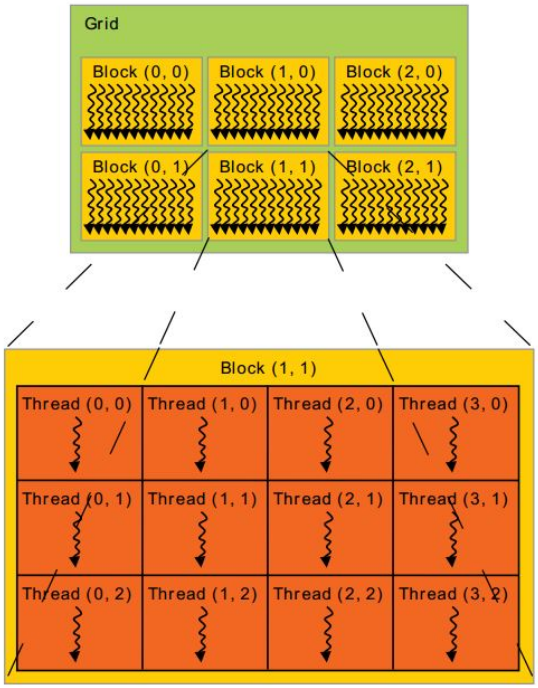
CUDA C rozširuje jazyk C tým, že umožňuje programátorom definovať funkcie - Kernely, ktoré sa pri zavolaní vykonajú N krát paralelne prostredníctvom N CUDA vlákien, narozdiel od klasickej C funkcie, ktorá sa vykoná iba raz.

Pri deklarovaní kernelu sa pred funkciu píše kľučové slovo `__global__`. Pre spustenie kernelu je ďalej potrebné nakonfigurovať počet vlákien v bloku a počet jednotlivých blokov. Na to slúži špeciálna syntax, písaná hneď za názvom volaného kernelu v tvare napr: `<<<1, N>>>`, prvý parameter konfiguruje počet blokov, ktoré budú kernel vykonávať. Druhý parameter nastavuje počet vlákien v jednotlivých blokoch. V príklade sa kernel vykoná jedným blokom s N vláknami. Pri zložitejších algoritmoch, prípadne pre prirodzenejšie adresovanie je možné konfigurovať vlákna v bloku až v troch rozmeroch a podobne aj bloky môžu byť usporiadané trojrozmerne. Pre správne adresovanie a rozlišovanie vlákien v blokoch poskytuje CUDA vstavanú premennú `threadIdx` ktorá má tri zložky x, y a z, cez ktoré je možné určiť polohu vlákna v rámci bloku. Bloky majú podobne ako vlákna svoju vstavanú premennú `blockIdx`, ktorá je rovnako trojrozmerná. Ďalej sú k dispozícii `blockDim` a `gridDim`, vďaka ktorým je možné v kóde kernelu určiť

[4] Označenie systému využívajúceho dátový paralelizmus. Nad množstvom dát sú vykonávané rovnaké inštrukcie

globálny index vlákna v rámci celého gridu. Napríklad pre súradnicu x by bol výpočet nasledovný: `index = blockDim.x * blockIdx.x + threadIdx.x`.

Počet vlákien v jednom bloku ale nie je neobmedzený, pretože všetky musia vykonávať svoju prácu na jednom jadre grafického procesoru a zdieľajú obmedzenú kapacitu pamäti jadra. Na súčasných GPU je počet vlákien v bloku obmedzený na maximum 1024. Avšak celkový počet vlákien podieľajúci sa na výpočte môže byť vyšší a je rovný počtu vlákien v jednom bloku vynásobený počtom blokov.



Obrázek 1: Príklad rozvrhnutia blokov a vlákien (Zdroj: [2])

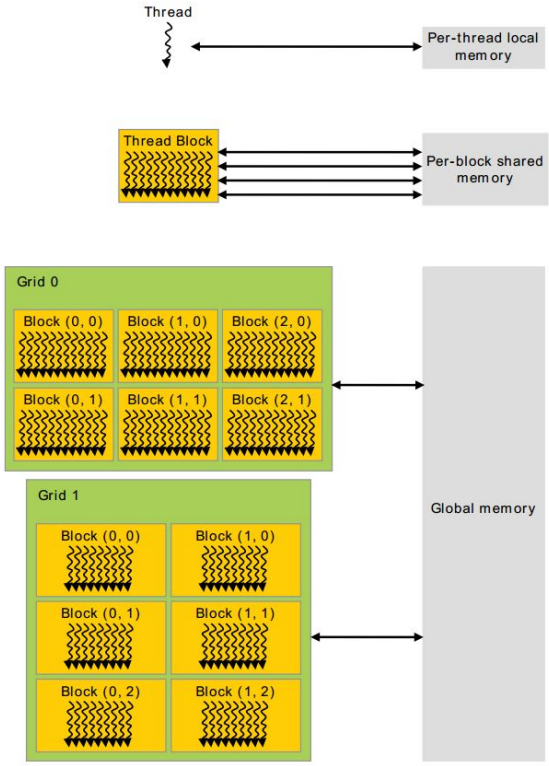
Vlákna dokážu medzi sebou zdieľať informácie v rámci rovnakého bloku prostredníctvom zdieľanej pamäte. Je to veľká výhoda najmä v aplikáciách, kedy sa pri výpočtoch zohľadňujú predchádzajúce výsledky a podobne. CUDA k tomuto prípadu poskytuje funkciu `__syncthreads()`, ktorá sa chová ako bariéra, ktorú nesmie prekročiť žiadne vlákno, pokiaľ sa k nej nedostanú všetky ostatné vlákna bloku.

Pre vysokú efektivitu kooperácie vlákien má zdieľaná pamäť veľmi nízku latenciu načo ju predurčuje jej fyzické umiestnenie blízko jadra procesoru. Podobne je aj spomínaná synchronizačná funkcia `__syncthreads()` nenáročná na počet inštrukcií.

4.2 Pamäť

Vlákna v CUDA kernely môžu pristupovať k dátam uloženým v rôznych typoch pamätí. Každá z nich má svoje špecifiká a hodí na sa určitý druh využitia. Každé vlákno má pre seba vyhradené registre, do ktorých sa ukladajú lokálne premenné vytvorené v kernely.

Bloky majú svoju zdieľanú pamäť, ktorej obsah má životnosť bloku, aby ju opäť mohol využiť iný blok. Všetky vlákna majú prístup do globálnej pamäte. CUDA ďalej poskytuje dve ďalšie podtypy pamäte, ktoré sú len na čítanie a ako globálna pamäť sú prístupné všetkým vláknam po celý čas: Pamäť na konštanty a pamäť na textúry. Tieto sú optimalizované na rôzne použitia, texturovacia naviac poskytuje rôzne adresovania a filtrovanie dát.



Obrázek 2: Diagram možností prístupov do pamäte v CUDA zariadení (Zdroj: [2])

4.2.1 Globálna pamäť

CUDA programovací model predpokladá, že systém je zložený z hosta[5] a zariadenia[6], kde každí má svoj vlastný oddelený pamäťový priestor. Kernel pracuje s pamäťou zariadenia, takže CUDA runtime poskytuje funkcie na jej alokovanie, dealokovanie, kopírovanie ako aj prenos dát medzi hostom a zariadením. Pamäť môže byť v CUDE alokovaná ako bežné lineárne pole, alebo CUDA pole líšiace sa rozvrhnutím pamäte prispôsobené na textúry.

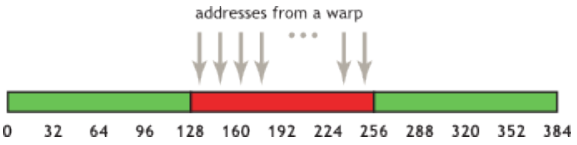
Lineárna pamäť sa alokuje funkciou `cudaMalloc`, pre viacrozmerné polia sú k dispozícii aj jej 2D a 3D verzie, naopak dealokácia je volaná cez funkciu `cudaFree`. Na prenos

[5] v CUDE je tak označovaný zvyšok systému bez grafickej karty
[6] grafická karta

medzi zariadením a hostom sa používa `cudaMemcpy`, ktorá dokáže kopírovať dáta v oboch smeroch.

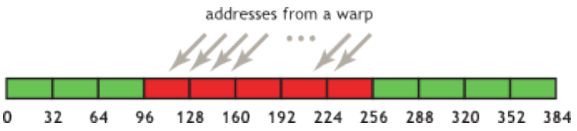
Globálna pamäť je uložená v ram pamäti zariadenia, preto má vysokú kapacitu. Využíva sa predovšetkým na prenos dát do zariadenia, kde už sa z nej používajú iba menšie časti.

Globálna pamäť nieje veľmi rýchla(v porovnaní so zdieľanou pamäťou), ale existujú techniky na zrýchlenie prístupu. V globálnej pamäti je možné využiť cache a splývanie pamäťových prenosov. Pre dosiahnutie splývania je potrebné aby vlákna jedného warpu[7] v bloku pristupovali do buniek globálnej pamäte v poradí ich indexu, v prípade že zároveň vlákna pristupujú do pamäte iba do jedného 128 bytového bloku dát a cacheovanie je povolené, prebehne táto operácia iba ako jedna pamäťová transakcia ako je to na obrázku 3. V prípade, že bude požiadavka zasahovať aj do iného bloku, vykonajú sa dve transakcie ako na obrázku 4. Bez použitia cache sa sa vykonajú iba 32 bytové transakcie, ich počet závisí na počte dát, podobne ako na obrázku 5.



Obrázek 3: prístup do globálnej pamäte využívajúci splývanie a cache - prístup cez jednu transakciu (Zdroj: [2])

Obrázek 4: sekvenčný, ale posunutý prístup do globálnej pamäte využívajúci splývanie a cache - prístup cez dve transakcie (Zdroj: [2])



Obrázek 5: sekvenčný - posunutý prístup do globálnej pamäte využívajúci splývanie bez cache - prístup cez päť transakcií (Zdroj: [2])

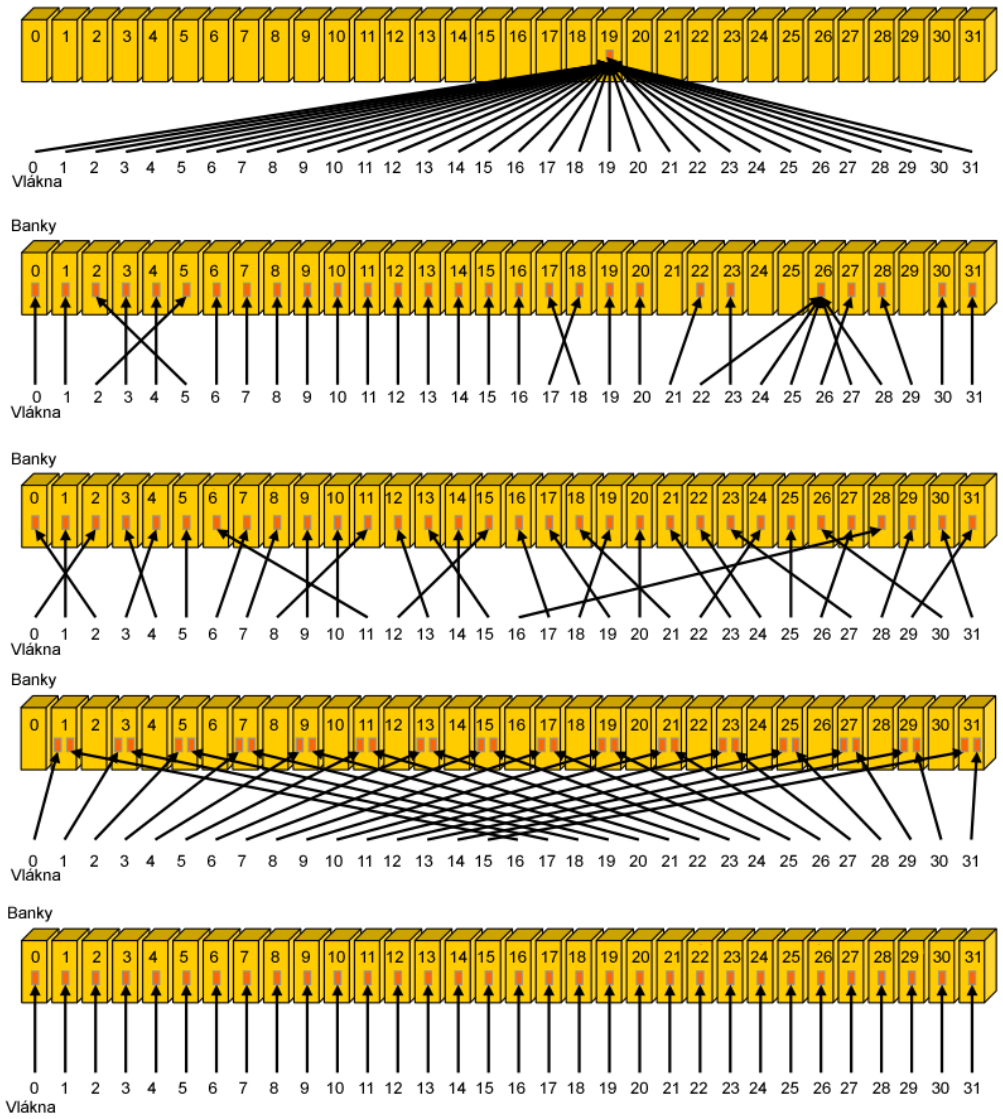
4.2.2 Zdieľaná pamäť

Pretože sa nachádza priamo na čipe je mnohonásobne rýchlejšia ako globálna pamäť, ako bolo spomínané v kapitole 4.1 za predpokladu, že sa vyhneme bankovým konfiktom.

[7] Nedeliteľná jednotka paralelizmu na CUDE, ktorá obsahuje 32 vlákien. Vlákna jedného warpu môžu vykonávať paralelne naplánované inštrukcie

Na dosiahnutie vysokej priepustnosti paralelných prístupov je zdieľaná pamäť rozdelená na rovnako veľké pamäťové moduly (banky), do ktorých je možné pristupovať paralelne. Takže každé načítanie alebo uloženie N dát do N rôznych bánk môže byť vykonané paralelne, z čoho sa dá odvodiť priepustnosť N krát vyššia ako priepustnosť jednej banky. V prípade, že viac vlákien pristupuje k jednej banke, prístup sa serializuje. Hardware rozdelí pamäťovú požiadavku, ktorá obsahuje bankový konflikt na požiadavky, ktoré už bankové konflikty neobsahujú. Tým sa priepustnosť znižuje toľko násobne, aký je počet nových bezkonfliktných požiadaviek. Existuje výnimka, kedy každé vlákno pristupuje k jedinej banke a prístup sa neserializuje a vlákna pristúpia k hodnote paralelne (broadkastovanie). V prípade novších typov grafických kariet (od compute capability[8] 2.x) je možné broadkastovať aj viac bánk medzi vlákna vo warpe, ako je ilustrované na obrázku 6.

[8]Verzia popisuje vlastnosti hardware a poskytuje informácie o inštrukčnej sade, ktorú zariadenie podporuje, ale aj špecifikácie ako maximálny počet vlákien v bloku alebo počet registrov v jadre procesoru. Zariadenia s vyšším číslom sú modernejšie, ale zároveň dovoľujú spätnú kompatibilitu so staršími verziami.



Obrázek 6: Prvý - bez konfliktu - vlákna pristupujú k rovnakým dátam v banke; Druhý - bez konfliktu - pretože medzi vlákna 22, 24, 25, 27 a 28 bude obsah banky broadcastovaný; Tretí - bez konfliktu - každé vlákno pristupuje k inej banke; Štvrtý - 2 cestný bankový konflikt - priepustnost'sa znižuje na polovicu; Piaty - bez konfliktu - každé vlákno pristupuje k inej banke. (Zdroj: [2])

5 Segmentácia obrazu

Segmentácia obrazu je metóda, alebo presnejšie skupina metód postavených na rôznych princípoch digitálneho spracovania obrazu, ktoré slúži k automatickému rozdeleniu vlastného obrazu na oblasti so spoločnými vlastnosťami a ktoré obvykle majú nejaký zmysluplný význam. Typickým cieľom segmentácie obrazu je identifikácia popredia a určenie oblastí v obraze odpovedajúcim významnému prvku zachytenej scény. Výsledky segmentácie sú využiteľné napríklad v počítačovom videní (rozpoznávanie tvárí, OCR[9] a iné), spracovaní lekárskych obrazových dát, alebo pri analýze obrazov získaných pri diaľkovom prieskume zeme [4].

5.1 Metódy segmentácie obrazu

V nasledujúcich riadkoch budú popísané niektoré metódy používané pri segmentovaní obrazu. Týchto metód je veľmi veľa, vybral som preto tie ktoré sú najpoužívanejšie.

5.1.1 Prahovanie

Prahovanie je najjednoduchšia metóda segmentácie obrazu založená na hodnotení jasu každého pixelu. Jej princípom je nájdenie takej hodnoty prahu, pre ktorú platí, že všetky hodnoty jasu nižšie ako prah patria do pozadia, zatiaľ čo vyššie hodnoty tvoria popredie.

5.1.2 Klastrovacie metódy

Do klastrovacích metód patrí aj metóda, ktorá je náplňou tejto práce - Spektrálne klastrovanie. Pri klastrovaní býva využívaný niektorý z klastrovacích algoritmov, ako bude spomenuté v kapitole 6.5. Samotné klastrovanie je založené na vybranej vlastnosti, ktorou býva farba, intenzita, textúra, poloha alebo iná vlastnosť dát.

5.1.3 Metódy založené na histograme

Ide o veľmi efektívne metódy v porovnaní s inými metódami segmentácie, pretože bežne vyžadujú iba jeden prechod poľom dát. Pri tomto prechode sa vypočíta histogram zo všetkých pixelov. Na určenie polohy segmentov/klastrov sa využívajú výrazné zmeny v priebehu histogramu ako sú vrcholy a priepasti.

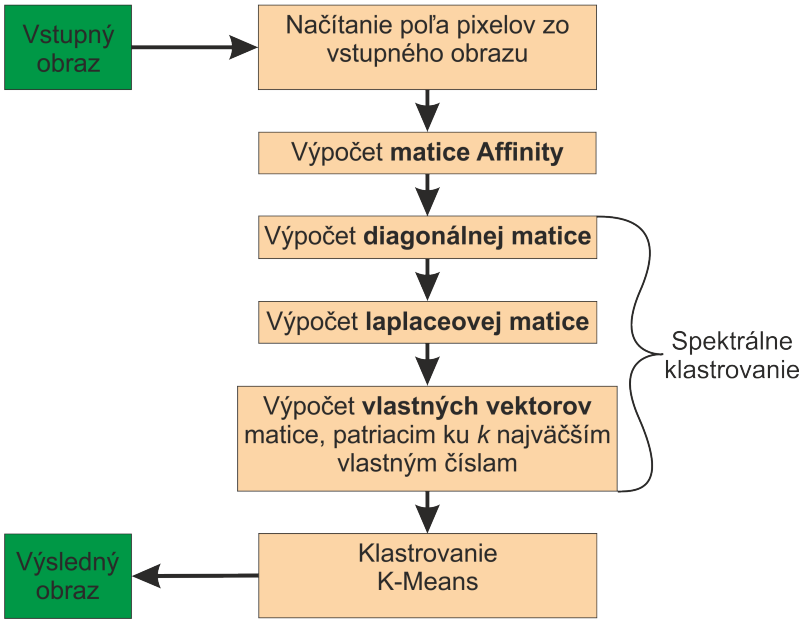
5.1.4 Metódy detekcie hrán

Detekcia hrán sa pri segmentovaní používa, pretože základnou vlastnosťou rozhrania objektu na popredí a pozadia je ostrá hrana. Preto sa detekcia hrán používa ako základ pre iné metódy segmentácie. V praxi sa využívajú gradientné operátory ako napríklad Cannyho hranový filter alebo Sobelov filter.

[9] Optické rozoznávanie znakov alebo OCR (z angl. Optical Character Recognition) je metóda umožňujúca preklad obrazu (grafiky) tlačených alebo písaných znakov do textovej, editovateľnej formy napr. do ASCII znakov abecedy.

6 Spektrálne klastrovanie

V tejto kapitole bude predstavené spektrálne klastrovanie. V prvom kroku sa načíta vstupný obraz v podobe R, G, B hodnôt jednotlivých pixelov. Zo vstupu sa vytvorí matica Affinity, ktorá vyjadruje vzájomnú podobnosť jednotlivých pixelov obrazu. Ak má vstupný obraz počet pixelov N, Affinity ako aj ostatné z nej vytvorené matice majú rozmer N^2. Z matice Affinity je vytvorená diagonálna matica, ktorej diagonálnymi prvkami sú umocnené súčty prvkov v riadkoch Affinity. Po výpočte diagonálnej a Affinity matice je možné ich násobením vypočítať Laplaceovu maticu. Ďalším krokom je výpočet k najväčších vlastných čísel a ich vlastných vektorov, ktoré sa uložia do matice ako stĺpce. Nasleduje klastrovanie bodov, ktoré sú reprezentované riadkami z matice vlastných vektorov. Po klastrovaní je ale potrebné premapovať riadky matice vlastných vektorov na obraz, ktorý je výsledkom tohto algoritmu. Tento postup znázorňuje obrázok 7. Jednotlivé kroky postupu budú popísané v nasledujúcich podkapitolách.



Obrázek 7: Graficky znázornený postup spektrálneho klastrovania.

6.1 Matica Affinity

V prvom kroku postupu je potrebné si vytvoriť maticu Affinity (ďalej len matica A). Matica A vyjadruje farebnú odlišnosť jednotlivých pixelov navzájom. Vstupný obraz je farebný, preto je farba pixelu tvorená farebnými zložkami, v tomto prípade zložkami RGB,

teda červenou, zelenou a modrou zložkou. pre výpočet farebnej odlišnosti používame nasledujúci vzťah [5]:

$$\sqrt{(X_r - Y_r)^2 + (X_g - Y_g)^2 + (X_b - Y_b)^2} \tag{1}$$

kde X je aktuálny pixel a Y je pixel s ktorým ho budeme porovnávať. Indexy r,g a b identifikujú o ktorú farebnú zložku pixelu ide. Jednotlivé zložky od seba odčítame, čo zaručuje nulovú odlišnosť pri pixeloch s rovnakou farbou a vyššiu odlišnosť pri pixeloch s farbou odlišnou. Vzťah je zhodný so vzťahom pre výpočet euklidovskej vzdialenosti v trojdimenzionálnom priestore, pretože v našom prípade ide o výpočet vzdialenosti farby pixelov vo farebnom spektre. Ak má vstupný obrázok šírku v pixeloch W a výšku v pixeloch H, matica A bude štvorcová a počet riadkov rovnako ako počet stĺpcov bude WH, pretože bude niesť odlišnosti kombinácie všetkých pixelov vstupného obrazu. Usporiadanie matice si dokážeme jednoducho predstaviť tak, že nad stĺpce prázdnej matice A si dáme farby pixelov vstupného obrazu usporiadené do rady a rovnako si ich dáme aj pred riadky prázdnej matice A. Pri výpočte ľubovoľného prvku matice by sme si ako pixel X dosadili hodnoty nad stĺpcom, v ktorom sa prvok nachádza a ako pixel B by sme si dosadili hodnoty pri riadku, v ktorom sa prvok nachádza. Pixely X a Y by sme si dosadili do vzťahu pre výpočet odlišnosti. Tento postup zopakujeme pre každý prvok matice A. Z tohto postupu je očividné, že prvky na diagonále budú nulové, pretože ide o odlišnosti pixelu porovnaného samého so sebou. Vzniknutá matica A je symetrická, to znamená že po transponovaní, teda po výmene riadkov za stĺpce, sa matica nijak nezmení. Je to spôsobené tým že pri porovnávaní pixelov, v našom príklade X s Y, budeme určite porovnávať aj Y s X, napríklad: porovnávame 1. pixel nad stĺpcami s 2. pixelom pri riadkoch, výsledný prvok bude v matici A uložený 1. stĺpci, 2. riadku. Neskôr budeme porovnávať 2. pixel nad stĺpcami s 1. pixelom pri riadkoch, výsledný prvok uložíme do 2. stĺpca a 1. riadku. Vypočítaná podobnosť bude v oboch prípadoch rovnaká, pretože ide o porovnanie iba dvoch pixelov. Výsledky sú uložené symetricky podľa diagonály, akurát si prvky vymenia pozície v riadku za stĺpec a naopak.

6.2 Diagonálna matica

Po zostavení matice A môžme vytvoriť diagonálnu maticu (ďalej len matica D). Matica D má rovnaké rozmery ako matica A. Ako už názov napovedá, jej prvky budú uložené na diagonále, všade inde budú mať prvky hodnotu nula. Pre výpočet prvkou využijeme vzťah [5]:

$$D_{ii} = \sum_{j=0}^{A_{width}} A_{ji} \tag{2}$$

D_{ii} je prvok na diagonále v matici D v i-tom stĺpci a i-tom riadku. Tento vypočítame ako sumu prvkov v i-tom riadku matice A. Matica D je rovnako ako A symetrická, čo je dané tým že je súmerná dodľa diagonály, pretože všade inde sú jej prvky rovné nule.

6.3 Laplaceová matica

Laplaceová matica (ďalej len matica L). V tomto bode je zrejmá podobnost's Laplaceovou maticou, ktorá je známa z grafov, na jej výpočet je potrebná matica nasýtenia, ktorá obsahuje 0 a 1 v závislosti na tom či majú dané uzly grafu medzi sebou hranu. Táto matica je veľmi podobná našej matici A. Ďalej je potrebná stupňová matica, ktorá opať ako v našom prípade obsahuje na diagonále sumy riadkov predchádzajúcej matice. Vďaka tejto podobnosti môžme na zostavenie matice L použiť vzťah pre výpočet symetrickej normalizovanej Laplaceovej matice [5]:

$$L_{norm} = D^{-1/2} L D^{-1/2} \tag{3}$$

kde $D^{-1/2}$ je matica D, v ktorej každý prvok na diagonále je rovný hodnote:

$$D_{ii} = \frac{1}{\sqrt{D_i i}} \tag{4}$$

Laplaceovu maticu ďalej využijeme pri výpočte vlastných čísel a vektorov.

6.4 Vlastné vektory

V tomto kroku si zostrojíme maticu V, ktorá bude obsahovať k najvačších vlastných vektorov matice L. Vlastné vektory môžme označiť $v_1, v_2, ...v_k$. Matica V bude teda vytvorená z vektorov $v_1, v_2, ...v_k$ ako jej stĺpcov. To znamená že prvé dimenzie vektorov budú v prvom riadku, druhé v druhom atď. Číslo k si volíme sami a je to počet klastrov do ktorých budeme vstupný obrázok deliť. Ak by sme zvolili napríklad $k = 4$, bude to znamenať že vo výsledku budeme obrazové body delit'do 4 skupín na základe ich podobnosti. Na základe čísla k bude algoritmus určovať toleranciu, ktorá bude zhlukovať obrazové body, ak bude k malé, napríklad $k = 2$, tolerancia bude vyššia aby bolo možné body rozdeliť do dvoch skupín a naopak, čím bude k vyššie bude tolerancia nižšia. Ďalej je potrebné maticu V normalizovať po riadkoch. Pri normalizácii využívame vzťah [5]:

$$V_{ij} = \frac{V_{ij}}{\sqrt{\sum_{j=0}^{V_{width}} V_{ij}^2}}; \tag{5}$$

kde V_{width} je šírka matice V. Zjednodušene povedané: sčítame druhé mocniny prvkov matice V v riadku, ktorý normalizujeme a následnú sumu odmocníme. Týmto výsledkom predelíme všetky prvky daného riadku, ktorý práve normalizujeme. Tým sa nám prvky v riadkoch - jednotlivé veľkosti vlastných vektorov v danej dimenzii numericky upravili do intervalu $\langle 0, 1 \rangle$.

6.5 Klastrovanie

Klastrovanie je úloha zaoberajúca sa zhlukovaním objektov, takže objekty v jednom klastry sú viac podobné (na základe vybranej metriky) medzi sebou ako ku objektom v iných klastroch.

Klastrovanie samo o sebe nie je jeden algortimus, ale vo všeobecnosti úloha, ktorá sa rieši pomocou rôznych algoritmov. Tie sa od seba môžu výrazne líšiť tým, ako definujú klaster a ako efektívne ho dokážu nájsť. Medzi známe metódy klastrovania patria: K-means, C-means fuzzy, Gaussian mixture, Single-Link a iné.

6.5.1 K-Means

Algoritmus zvolený k riešeniu problému sa nazýva k-means [6]. Tento algoritmus je veľmi populárny a často využívaný. Základnou myšlienkou je vytvorenie k centroidov, ktoré reprezentujú dátové body vo vnútri klastru (segmentu) S. Vyjadrene formálne, máme niekoľko pozorovaní $(x_1, ..., x_n)$, kde $x_j \in R^d$ a d predstavuje dimenziu každého z pozorovaní. Ďalej máme segmentáciu $S = \{S_1, ...S_k\}$, v ktorej chceme minimalizovať vnútroklastrový súčet umocnených vzdialeností, tak ako to vyjadruje vzťah

$$argmin_S = \sum_{i=1}^{k} \sum_{x_j \in S_i} \|x_j - \mu_i\|^2 \tag{6}$$

K-means sa k výsledku tejto funkcie dostáva použitím dvoch krokov:

- **Priraďovanie**, v ktorom sú všetky pozorovania priradené do jedného z klastrov, na základe ich vzájomnej vzdialenosti, teda vzdialenosti pozorovania a centroidu klastru.
- **Aktualizácia**, v ktorom sú centroidy posunuté na novú pozíciu reprezentujúcu novú strednú hodnotu priradených dátových bodov (pozorovaní). Tieto dva kroky môžu byť vyjadrené nasledujúcimi rovnicami.

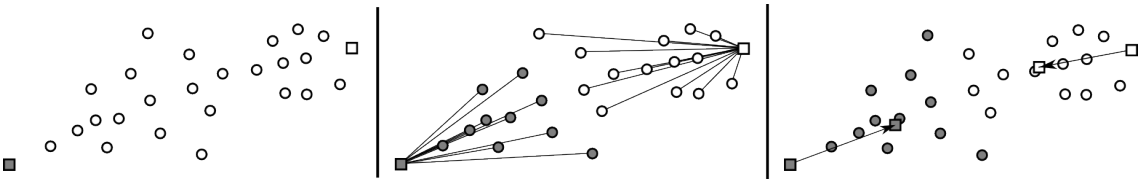
$$S_i^{(t)} = \{x_a : \|x_a - \mu_i^{(t)}\|^2 \leq \|x_a - \mu_j^{(t)}\|^2 \forall j, 1 \leq i \leq k\} \tag{7}$$

$$\mu_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j \tag{8}$$

kde x_a je dátový bod, ktorý chceme priradiť k jednému z klastrov (segmentov).

Problémom algoritmu k-means je jeho zložitosť (klasifikovaný ako NP-ťažký problém) a nezaručená konvergencia ku klastrom, v prípade že sa použije iná metrika ako Euklidovská. Na nasledujúcom obrázku je znázornený proces klastrovania algoritmom k-means.

Našimi pozorovaniami sú práve jednotlivé riadky v normalizovanej matici V. Matica V má k stĺpcov, každý tento stĺpec si teraz predstavíme ako jednu dimenziu bodu - riadku. Môže to byť trochu mätúce, pretože pri konštrukcii matice V to bolo presne naopak a vektory z ktorých sa vytvorila sa poukladali vedľa seba ako stĺpce. Máme teda maticu V s k stĺpcami a počtom riadkov, rovným celkovému počtu obrazových bodov vstupného obrazu. Táto informácia už možno naznačuje, že medzi vstupným



Obrázek 8: Schéma klastrovania prostredníctvom k-means. Prvý obrázok: inicializácia centroidov; druhý obrázok: priradenie bodov k centroidom; tretí obrázok: aktualizácia centroidov;

obrazom a maticou V je spojitosť. Po priebehu k-means algoritmu budeme vedieť do akých skupín - zhlukov sa jednotlivé riadky z matice V zaradili a podľa toho budeme vedieť aj príslušnosť jednotlivých obrazových bodov vstupného obrazu do zhlukov. Povedzme že máme obrázok 2 krát 2 pixely, máme maticu V, ktorá má 4 riadky (ako celkový počet pixelov) a 2 stĺpce (hodnota k). Budeme teda rozdeľovať 4 body v 2 rozmernom priestore do 2 zhlukov. Po spustení algoritmu zistíme, že 1. a 3. bod sa nachádza v 1. zhluku a 2. a 4. bod v zhluku druhom. Pri namapovaní na vstupný obrázok: 1. pixel v 1. a 2. riadku bude označený jednou farbou a 2. pixel v 1. a 2. riadku farbou inou, pretože patrí do druhého zhluku.

Matica V:
$$\begin{array}{c} \\ x \\ o \\ x \\ o \end{array} \begin{array}{c} \begin{array}{cc} x & y \end{array} \\ \begin{pmatrix} a & b \\ c & d \\ e & f \\ g & h \end{pmatrix} \end{array}$$

Premapovaný výstupný obraz:
$$\begin{array}{c} \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{cc} 1 & 2 \end{array} \\ \begin{pmatrix} x & o \\ x & o \end{pmatrix} \end{array}$$

7 Návrh riešenia

Táto časť práce sa venuje praktickej implementácii algoritmu spektrálneho klastrovania, popísaného v kapitole 6. Algoritmus bol implementovaný najskôr sekvenčne s použitím procesoru, neskôr paralelne s použitím technológie CUDA. Obe verzie budú v tejto kapitole popísané.

7.1 Knižnica OpenCV

OpenCV je súbor knižníc, ktorý obsahuje funkcie pre manipuláciu s obrazom. V prvých verziách sekvenčného riešenia bolo použité násobenie matíc a aj výpočet vlastných čísel a vlastných vektorov matice. V poslednej verzii algoritmu bolo použité už len načítanie obrazu cez funkciu `cv::imread`, ktorá načíta obraz zo súboru do matice v hodnotách typu unsigned char[10] a vracia instanciu triedy `cv::Mat` cez ktorú sa už dá jednoducho pristupovať priamo k hodnotám zložiek farby jednotlivých pixelov. Po skončení klastrovania je použitá funkcia `cv::imshow` do ktorej sa parametrom predá odkaz na maticu výsledných obrazových bodov a funkcia daný výsledok zobrazí. Verzia použitá pri vývoji algoritmov bola 2.4.6.

7.2 Sekvenčné riešenie

Z názvu kapitoly vyplýva, že popisovaný algoritmus bol implementovaný jednovláknovo na procesore. Oproti algoritmu na CUDE je technické riešenie takmer úplne odlišné, aj keď postup je veľmi podobný. Sekvenčné riešenie je oproti CUDE omnoho jednoduchšie, pretože pri iterovaní jednotlivých výpočtov, napríklad matice L je dopredu známe a postupné a odpadá teda nutnosť prepočítavania indexov v matici, alebo vo vektoroch.

7.2.1 Naivné riešenie

Sekvenčný algoritmus rovnako ako ten paralelný prešiel veľa štádiami. V prvotnom tzv. naivnom riešení bol implementovaný algoritmus popisovaný v kapitole 6 bez výraznejších zmien. Pri výpočte matice A sa teda vytvorí matica o veľkosti počtu pixelov vstupného obrazu umocnených na druhú. Tu je zrejmé, že aj z hľadiska rýchlosti a hlavne z hľadiska priestorovej náročnosti to nie je dobré riešenie. V matici A sú totiž informácie o rozdieloch obrazových bodov ukladané duplicitne. Podobne to je aj s vytváraním matice D, ktorá obsahuje prvky iba na diagonále, avšak matica je alokovaná celá o rovnakej veľkosti ako matica A. Vytváranie matice L súčinom matíc:

$$L_{norm} = D^{-1/2} A D^{-1/2} \tag{9}$$

Tento výpočet je realizovaný cez knižnicu OpenCV, ktorá má pre instancie `cv::Mat` vlastný operátor násobenia. Toto riešenie je absolútne neefektívne, pretože výpočet prebieha aj nad nulovými prvkami matice D, kedy je výsledok nenulový iba v jednej odmocnine z celkového počtu všetkých násobení.

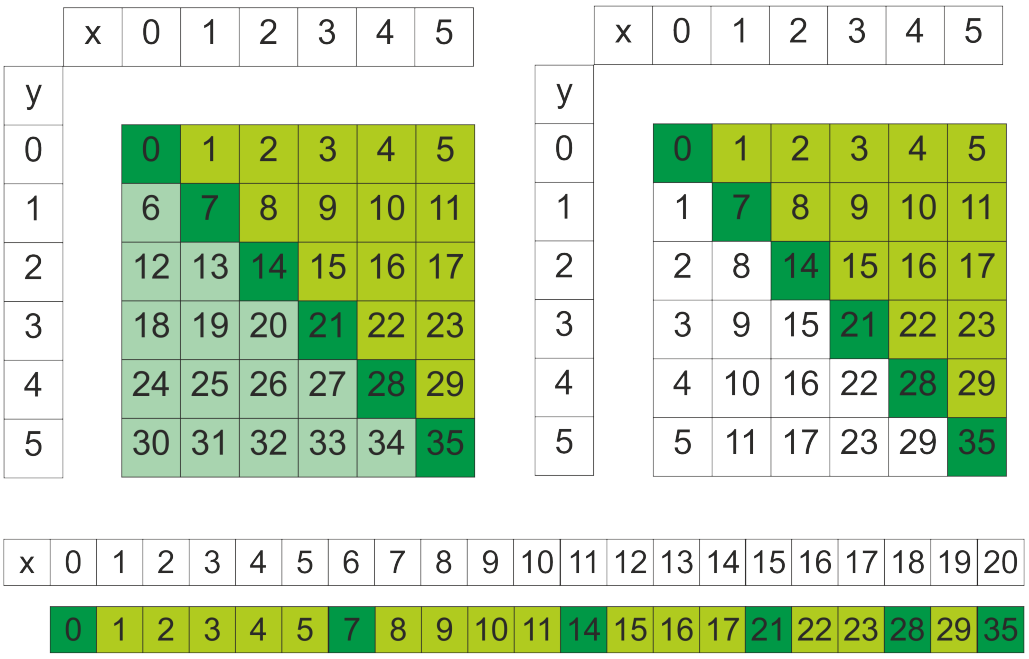
[10] jednobytová premenná bezznamienkového typu, nadobúdajúca hodnoty v intervale [0,255]

Výpočet vlastných vektorov matice je realizovaný opäť cez OpenCV, cez funkciu `cv::eigen`. Pri nízkych rozlíšeniach je tento výpočet prijateľný, ale pri väčších maticiach je opäť neefektívny, pretože funkcia vypočíta všetky vlastné čísla, rovnako ako vektory. Preto je potrebné vypočítať iba k vlastných čísel a vlastných vektorov.

7.2.2 Vylepšené riešenie

Tento algoritmus je konečným sekvenčným algoritmom implementovaným vrámci tejto práce. Snaží sa eliminovať nedostatky naivného riešenia, ktoré boli spomínané v predchádzajúcej kapitole 7.2.1.

7.2.2.1 Matica Affinity Ako prvý nedostatok bolo potrebné vyriešiť efektívne uloženie matice Affinity (ďalej matica A). Keďže je matica symetrická, stačí uložiť iba jej trojuholníkovú časť vrátane diagonály. Ak by matica nebola symetrická[11], nebolo by ju možné takto ukladať. Preto bolo potrebné zvoliť formát v ktorom budú dáta uložené, aby bolo možné k prvkom určiť indexy na ktorých boli dáta uložené v pôvodnej plnej matici, kvôli výpočtu daného prvku matice z hodôt pixelov vstupného obrazu. V sekvenčnom algoritme bolo zvolené ako ideálne uloženie v lineárnom poli, tak že sú riadky uložené za sebou. Každý nasledujúci riadok má o jeden prvok menej ako jeho predchodca.



Obrázek 9: Usporiadanie poľa hodnôt horného trojuholníku matice.

Na obrázku 9 napravo je ilustrovaná plná matica (pre názornosť je matica veľkosti 6x6), na ktorej sú farebne vyznačené oba trojuholníky. Na pravo je zobrazená zreduko-

[11] napríklad pri využití inej metriky pri vytváraní zo vstupného obrazu

vaná matica, už bez spodného trojuholníku. Nižšie je ilustrovaný princíp uloženia do jednorozmerného poľa.

Pre alokovanie pamäte je potrebné vedieť počet dát, ktoré bude obsahovať zredukovaná matica. Tento výpočet je možné si odvodiť: potrebujeme miesto pre horný trojuholník. Rovnaký trojuholník je aj pod diagonálou. Ak teda predelíme celkovú veľkosť matice na polovicu, získame miesto pre trojuholník a polovicu diagonály, preto je potrebné ešte zahrnúť aj druhú polovicu diagonály. A to tak, že k výsledku pričítame polovicu počtu stĺpcov matice. Celý vzorec je teda nasledovný:

$$Size = \frac{m^2}{2} + \frac{m+1}{2} \tag{10}$$

Vo vzorci m vyjadruje jeden z rozmerov matice. V druhej časti je ešte k hodnote m pričítaná jednotka. Je to z toho dôvodu, že pri počítaní veľkosti pri matici, ktorej rozmery sú nepárne by výpočet algortimu zaokrúhlil obe delenia smerom nadol, pretože v programe je výpočet robený nad typmi integer[12]. Z toho dôvodu je nutné k výpočtu pripočítať číslo jedna, čo zaručí, že pri akejkoľvek matici bude výpočet zaokrúhľovať iba jeden z dvoch zlomkov.

Hodnoty pôvodných indexov sa dajú ľahko určovať tak, ako je to v nasledujúcom kóde, ktorý je ukážkou vytvorenia matice A zo vstupného obrazu:

```
float * getS(cv::Mat &image)
{
    int m = image.cols * image.rows;
    int i = 0;
    float * S = (float*)malloc(((m * m + m + 1)/2) * sizeof(float));

    cv::Vec3b a,b;
    for (int y = 0; y < m; y++){
        for (int x = y; x < m; x++){

            a = image.at<cv::Vec3b>( GETY(y,image.cols), GETX(y,image.cols) );
            b = image.at<cv::Vec3b>( GETY(x,image.cols), GETX(x,image.cols) );
            S[i] = ES(a,b);

            i++;
        }
    }
    return S;
}
```

Výpis 1: Vytvorenie matice Affinity

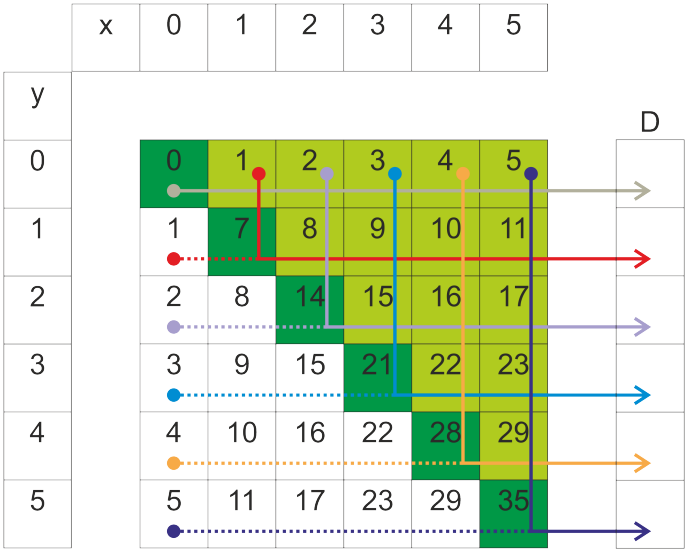
Cyklus cez y prechádza riadky nezredukovanej matice, cyklus cez x prechádza stĺpce, avšak vždy začína až na diagonále, čo zaručí iba počet iterácií rovný počtu prvkov zredukovanej matice. Pre prístup k prvkom obrázku je potrebné upraviť indexy x a y z jednorozmerných na dvojrozmerné, pretože aj vstupný obraz je uložený ako dvojrozmerné pole. Makro `GETX(a,b)` počíta zvyšok po delení a/b a zaisťuje index v stĺpci tým,

[12] 4 bytová premenná celočíselného typu.

že s rastom hodnoty a sa rovanko zvyšuje aj výsledná hodnota až do hodnoty b, kde sa výsledok opäť dostane na hodnotu 0 atď. `GETY(a,b)` počíta celočíselné delenie a/b a určuje riadok. Premenná i je ako inkrementer, zaisťujúci postupný posun vo výslednej matici A.

Tento prístup eliminuje zbytočné duplicitné dáta a rovnako aj duplicitné výpočty. Obe spomínané zníži takmer na polovicu.

7.2.2.2 Diagonálna matica Pri výpočte diagonálnej matice (ďalej len matica D) je potrebné sčítať prvky v riadkoch matice A. Akonáhle matica A obsahuje iba horný trojuholník nastáva problém, kde treba premapovať prvky z dolného trojuholníku matice na ten horný. V tomto prípade opäť nemá zmysel prechádzať veľkosť celej nezredukovanej matice A. Pre výpočet stačí prejsť iba horný trojuholník matice podobne ako pri výpočte matice A. V prvom riadku je všetko v poriadku a prvky stačí sčítať. V druhom riadku už chýba prvý prvok, jeho súradnice sú $x = 0$ a $y = 1$, ale jeho duplicita sa nachádza na tom istom mieste v transponovanej matici. To znamená, že stačí súradnice zamenit' a dostávame indexy v pôvodnej matici v hornom trojuholníku.



Obrázek 10: Premapovanie prvkov dolného trojuholníka matice na horný.

Pre lepšiu predstavu poslúži opäť ilustračný obrázok 10, ktorý znázorňuje premapovanie dolného trojúholníku na horný. Každý prvok matice, cez ktorý prechádza šípka sa do súčtu musí prirátať. Pri implementácii je možné prejsť cyklus cez celú nezredukovanú maticu A a podľa toho, či je aktuálny index cyklu pod alebo nad diagonálou zamieňať indexy x a y. Celé rozhodovanie je ale možné eliminovať, takže cyklus prejde iba horný trojuholník vrátane diagonály a pri každom cykle bude sčítavať daný riadok do matice (vektoru) D na pozíciu y - v obrázku znázorňujú vodorovné šípky, zároveň daný prvok pripočíta k D na pozíciu x - v obrázku zvislé šípky. Ako bolo poznačené vyššie, D nieje ukladané v matici, ale vo vektore kvôli šetreniu miesta.

Samotný výpočet v sebe ešte zahŕňa umocnenie pričítavaného prvku. Po skončení výpočtu súm sa prvky vektoru D prepočítajú na obrátené hodnoty ich odmocnín ako popisuje vzťah 4.

7.2.2.3 Laplaceova Matica Výpočet Laplaceovej matice bol v popisovaný v kapitole 6.3. Tam ale algoritmus počíta s tým že D je matica, ale v tomto prípade ide o vektor. Preto je potrebné upravit' výpočet tak aby výsledok odpovedal násobeniu matíc, ako je to v spomínanom vzťahu.

Ak si zoberieme diagonálnu maticu[13] a vynásobíme ju s inou maticou a výslednú maticu opät' vynásobíme s danou diagonálnou maticou, vyjde nám niečo podobné:

$$\begin{aligned}\begin{pmatrix} d_{11} & 0 & 0 \\ 0 & d_{22} & 0 \\ 0 & 0 & d_{33} \end{pmatrix} \begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{pmatrix} \begin{pmatrix} d_{11} & 0 & 0 \\ 0 & d_{22} & 0 \\ 0 & 0 & d_{33} \end{pmatrix} \\ = \begin{pmatrix} d_{11}x_{11}d_{11} & d_{11}x_{12}d_{22} & d_{11}x_{13}d_{33} \\ d_{22}x_{21}d_{11} & d_{22}x_{22}d_{22} & d_{22}x_{23}d_{33} \\ d_{33}x_{31}d_{11} & d_{33}x_{32}d_{22} & d_{33}x_{33}d_{33} \end{pmatrix}\end{aligned} \tag{11}$$

Pri preštudovaní výsledku si môžme všimnút', že riadky matice s prvkami x sú prenásobené diagonálou, rovnako ako aj stĺpce. Preto je uloženie D vo vektore absolútne prijateľné a práve naopak viac vyhovujúce. A pretože sa vo výpočte nepočíta žiadna suma a po skončení by symetrická matica ostala stále symetrickou, je možné počítat' hodnoty iba pre našu zredukovanú maticu.

```
void getL(float ** A, cv::Mat & D)
{
    int i = 0;
    for (int y = 0; y < D.cols; y++){
        for (int x = y; x < D.cols; x++){
            (*A)[i] =
                D.at<float>(0,y) *
                (*A)[i] *
                D.at<float>(0,x);

            i++;
        }
    }
}
```

Výpis 2: Vytvorenie Laplacianovej matice

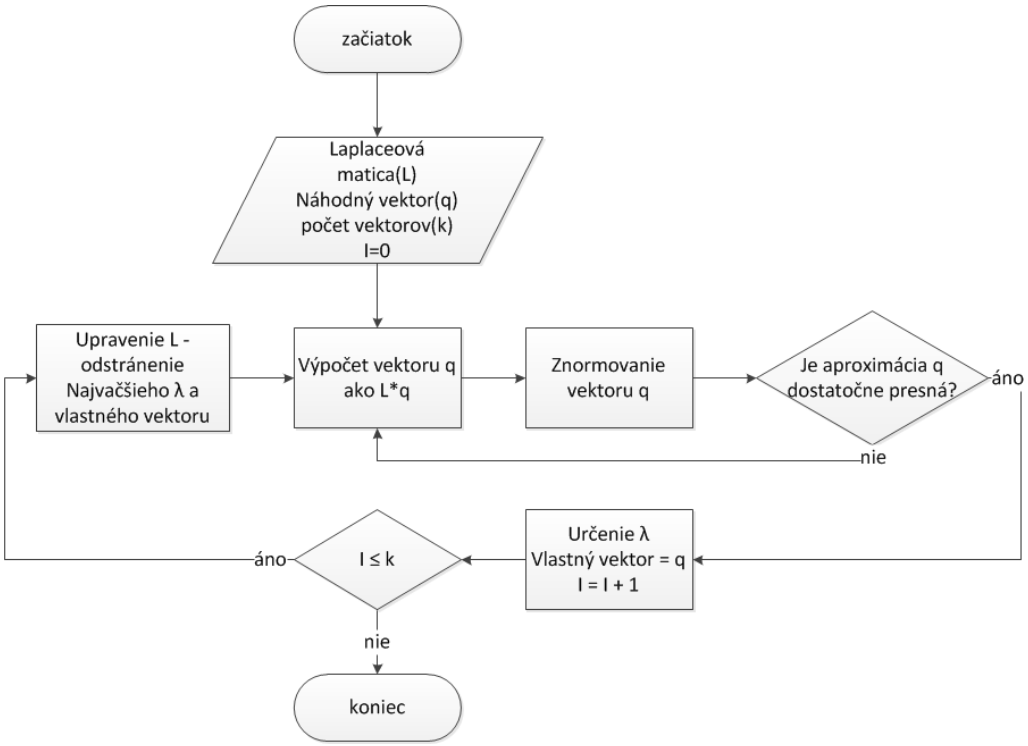
Kód je podobný tomu na výpočet matice L. `D.cols` je dĺžka vektoru D, rovnaká ako oba rozmery nezredukovanej matice A.

[13] Matica, ktorá má nenulové prvky iba na diagonále.

7.2.2.4 Výpočet vlastných vektorov Na výpočet vlastných vektorov Laplacianovej matice bol zvolený algoritmus Power Iteration. Ide o metódu, ktorá zo zadanej matice L vyprodukuje vlastné číslo λ a nenulový vlastný vektor v tak že platí:

$$Lv = \lambda v \tag{12}$$

Metóda nepočíta rozklad matice, preto je vhodná aj na obrovské matice, takže je veľmi vhodná do tohto algoritmu. Základná metóda Power Iteration nájde iba jedno najväčšie vlastné číslo (s najvačšou absolútnou hodnotou). Ďalšími úpravami je ale schopná vypočítať aj ďalšie vektory a čísla.



Obrázek 11: Diagram znázoňujúci postup krokov metódy Power Iteration.

Predpokladajme že matica $L \in R^{n \times n}$ je diagonalizovateľná[14], to znamená že existuje n nezávislých vlastných vektorov matice L. Označme tieto vektory ako $x_1, ...x_n$, tieto vektory tvoria bázu priestoru R^n. Zvolíme si náhodný vektor $q^{(0)} \in R^n$, ktorý môžme zapísať aj takto[7]:

$$q^{(0)} = a_1x_1 + a_2x_2 + ... + a_nx_n \tag{13}$$

kde $a_1, ...a_n$ sú skaláre. Rovnica hovorí, že pomocou lineárnej kombinácie vlastných - bázových vektorov je možné vytvoriť akýkoľvek iný vektor.

[14]Matica L je diagonalizovateľná ak k nej existuje matica X tak že platí: $L = XDX^{-1}$ a D je diagonálna matica.

Vynásobením oboch strán rovnice maticou L umocnenou na k dostávame:

$$L^k q^{(0)} = L^k(a_1 x_1 + a_2 x_2 + ... + a_n x_n) \tag{14}$$

Po roznásobení dostávame:

$$= a_1 L^k x_1 + a_2 L^k x_2 + ... + a_n L^k x_n \tag{15}$$

Zo vzťahu 12 nám vyplýva, že platí $L^k x_n = \lambda^k x_n$. Preto môžme v rovnici maticu L nahradiť vlastnými číslami $\lambda_1, ...\lambda_n$:

$$= a_1 {\lambda_1}^k x_1 + a_2 {\lambda_2}^k x_2 + ... + a_n {\lambda_n}^k x_n \tag{16}$$

Teraz je možné si pred zátvorku vyňať prvý skalár a najvačšie vlastné číslo $a_1 {\lambda_1}^k$. Zvyšok prepíšeme do formy sumy, z ktorej je potrebné vydeliť to čo sme vyňali aby ostala rovnica ekvivalentná:

$$= a_1 \lambda_1^k \left(x_1 + \sum_{j=2}^{n} \frac{a_j}{a_1} \left(\frac{\lambda_j}{\lambda_1} \right)^k x_j \right) \tag{17}$$

Rýchlosť Power Iteration je závislá na pomere $|\lambda_2|/|\lambda_1|$, ktorý určuje rýchlosť konvergencie. Čím je teda pomer menší, tým skôr sa suma zo vzťahu 17 začne blížiť 0. Z experimentálne vypočítaných vlastných čísel z obrázkov sa preukázalo, že počet výrazne odlišných vlastných čísel bol úmerný počtu výrazne odlišiteľných farieb v obraze. Preto nevzniká problém s tým, že by algoritmus konvergoval pomaly, alebo dokonca vôbec.

Ak teda platí $|\lambda_1| > |\lambda_2| \geq ...|\lambda_n|$ potom môžme povedať že λ_1 je dominantné vlastné číslo matice L. Z predpokladu, že každé ďalšie vlastné číslo bude menšie ako dominantné, môžme odvodiť $\left(\frac{\lambda_j}{\lambda_1}\right)^k \to 0$. To znamená, že čím vyššie bude číslo k, tým sa bude celá suma blížiť viac k nule. A za predpokladu $a_1 \neq 0$ môžme teda odvodiť princíp algoritmu Power Iteration:

$$L^k q^{(0)} \to a_1 \lambda_1^k x_1 \tag{18}$$

Jednotlivé iterácie môžu byť vyjadrené rovnicami:

$$\begin{aligned} q_1 &= Lq_0 \\ q_2 = Lq_1 &= L(Lq_0) = L^2 q_0 \\ q_3 = Lq_2 &= L(L^2 q_0) = L^3 q_0 \\ &\vdots \\ q_k = Lq_{k-1} &= L(L^{k-1} q_0) = L^k q_0 \end{aligned} \tag{19}$$

Metóda pri každej iterácii vektor po násobení Lq^{k-1} normalizuje, aby pri výpočtoch nad veľkými maticami, hodnoty výsledného vektoru nepretiekli dátový typ. Vďaka

normalizácii teda produkt konverguje priamo k vlastnému vektoru. Normalizácia je vyjadrená vzťahom:

$$x_i = \frac{x_i}{\sqrt{\sum_{j=0}^{n} x_j}} \tag{20}$$

Znormalizovaný vektor q každou iteráciou viac aproximuje k hľadanému vlastnému vektoru. Ale keďže počet iterácií, ktorý vyprodukuje uspokojivý výsledok nieje známy, je potreba stanovit' ukončovaciu podmienku. V tomto prípade sa dá ako ukazateľ presnosti použit' rozdiel medzi vektormi vypočítanými v dvoch po sebe idúcich iteráciách. Podmienku zastavenia vyjadríme ako

$$\|q_k - q_{k-1}\| < tolerancia \tag{21}$$

kde q_k je aktuálny znormovaný vektor a q_{k-1} je znormovaný vektor z predchádzajúcej iterácie. Tolerancia je numerická hodnota, ktorá predstavuje hodnotu prípustného rozdielu. Experimentovaním sa ukázalo, že vektor konverguje veľmi rýchlo a aproximácia je už po pár iteráciách veľmi presná. Vo výsledku sa ukázala hodnota tolerancie $1 * 10^{-4}$ ako úplne dostačujúca a jej znižovanie už nemalo žiadny vplyv na výsledný vlastný vektor.

Hľadanie nedominantných vlastných čísiel. V klastovaní jeden vlastný vektor väčšinou nie je dost', preto je potreba modifikovat' algoritmus, aby bol schopný vypočítat' aj ďalšie vlastné vektory. K ďalšiemu vektoru potrebujeme vlastné číslo prvého vektoru. Výpočet je definovaný ako [8]:

$$\lambda_i = \frac{Lx_i x_i}{x_i x_i} \tag{22}$$

x_i je i-ty vypočítaný vlastný vektor. Zdrojový kód výpočtu vlastného čísla:

```
float getEigVal( cv::Vec & eigvec, cv::Vec & qvec )
{
    float eigval = 0;
    for( int i = 0; i < qvec.size; i++)
        eigVal +=   qvec[i] * eigvec[i];

    return eigval;
}
```

Výpis 3: Výpočet vlastného čísla

Násobenie matice L a vektoru x_i produkuje vektor, no ten nieje potreba počítat' pretože už z výpočtov iterácií to je vypočítaný neznormovaný vektor q. Podobne netreba ani delenie produktom vektorov $x_i x_i$, pretože ide v podstate o normovanie, avšak vypočítaný vlastný vektor už normovaný je.

Výsledné vlastné číslo je ale v absolútnej hodnote. Číslo λ je kladné, ak majú zložky normovaného vektoru x_i na rovnakých pozíciach rovnaké znamienka ako x_{i-1}. V prípade že sa znamienka každou iteráciou menia je číslo λ záporné [9].

Za predpokladu že matica L je symetrická môžme ju rozložiť na:

$$L = X\Lambda X^T \tag{23}$$

Kde X a Λ obsahujú vlastné čísla a vektory. Tento vzťah môžme upraviť do podoby obsahujúcej priamo vlastné čísla a vektory:

$$L = \sum_{i=1}^{n} \lambda_i x_i x_i^T \tag{24}$$

Na základe tohto vzťahu sme schopní vytvoriť novú maticu, ktorá už nebude obsahovať najvačšie vlastné číslo a vektor. Tento postup sa nazýva deflácia.

$$\hat{L} = L - \lambda_1 x_1 x_1^T \tag{25}$$

Matica $\hat{L}$ obsahuje rovnaké vlastné čísla a aj vektory okrem dominantného. Tento fakt zaručuje možnosť vypočítať aj nedominantné vlastné čísla. Ak teda upravíme vstupnú maticu L podľa tohto vzťahu, môžme nad ňou opäť vykonať power iteration a dostaneme sa ku ďalšiemu vlastnému číslu a vektoru. Tento postup opakujeme podľa počtu potrebných vlastných vektorov. Pri ďalších opakovaniach sa ale odčítavajú posledné vlastné čísla a vektory. Ak by sme počítali napríklad tretie najvačšie číslo, z matice už musia byť odpočítané prvé dve najvačšie a aj ich vektory.

V praktickom riešení používa aj metóda na výpočet vlastných vektorov a vlastných čísiel usporiadanie matice L v pamäti rovnaké ako predchádzajúce metódy na vytvorenie matice A, D a aj L. Je tak opäť ušetrené miesto. Spôsob indexovania a prístupov do pamäte bol popísaný v podkapitolách Matica A 7.2.2.2, Matica D 7.2.2.2.

Po výpočte máme maticu, v ktorej sú ako stĺpce jednotlivé vlastné vektory. Túto maticu označíme ako V.

7.2.2.5 Klastrovanie Ako bolo spomínané v kapitole 6, v riešení je použitá na klastrovanie metóda K-means. Teoretické pozadie algoritmu bolo popísané v podkapitole 6.5.1.

Ako prvé je potrebné si zvoliť centroidy pre každý z klastrov. Centroidov ako aj klastrov bude k. Centroidy si môžme voliť náhodne, tak že si vygenerujeme náhodný bod v k-dimenzionálnom priestore a ten označíme ako centroid prvého zhluku a takto pokračujeme ďalej. Avšak hodnota k za bežných podmienok nebude nijak vysoká ale zostane v rádoch jednotiek. Preto úplne postačí ak si za centrá zvolíme prvých k bodov - riadkov z matice V. V ďalšom kroku algoritmu prejdeme všetky body matice V a spočítame vzdialenosť od daného bodu ku každému centroidu [5].

$$\sqrt{\sum_{i=1}^{k} q_i - {p_i}^2} \tag{26}$$

Kde q a p sú body v k-dimezionálnom priestore. Za jeden bod si dosadíme skúmaný bod a za druhý centroid. Pre každý centroid si pre daný bod zistíme vzdialenosť a najkratšia nám určuje, že daný bod patrí do klastru s týmto centroidom. Po prechode

všetkými bodmi matice V máme body zatriedené do klastrov. Opäť ideme zistiť nové centroidy klastrov, tentoraz ale tak že si vypočítame centroid medzi bodmi daného zhluku (vektorový zápis) [5]:

$$C = \frac{1}{n} \sum_{i=0}^{n} x_i \tag{27}$$

kde x_i je i-ty prvok v danom klastry, n je celkový počet prvkov v klastry C je nová poloha centroidu. Po vypočítaní jednotlivých nových centroidov, ich porovnáme s pôvodnými centroidmi a ak sa aspoň jeden zmenil, je potrebné celý postup zopakovať. To znamená opäť prejsť všetky body a zatriediť ich, opäť vypočítať centroidy atď. V prípade, že sa staré a nové centroidy (vypočítané) nelíšia je algoritmus k-means na konci.

Ako najoptimálnejšie sa ukázalo riešenie, kde sa vytvorí vektor s rozmerom ako vlastné vektory, ktorý bude ukladať informáciu o príslušnosti daného bodu k jednému z k-klastrov. Čiže vektor bude nadobúdať hodnoty z intervalu $[0, k)$.

7.2.2.6 Premapovanie Poslednou časťou je premapovanie vektoru, ktorý nám vygeneroval k-means na obraz. Podobne ako bolo spomínané v kapitole 7.2.2.1, sa prepočíta jednorozmerný index na indexy riadku a stĺpcu v obrázku. Teraz stačí hodnoty vektoru z intervalu $[0, k)$, prepočítať na hodnoty z intervalu $[0, 255]$ aby boli vo výsledku príslušnosti bodov do klastrov dobre rozlišiteľné. Stačí aby sme pôvodnú hodnotu prenásobili $255/(k-1)$ čo hodnoty rovnomerne rozloží na celý interval.

7.3 Paralelné riešenie

Paralelné riešenie rovnako ako sekvenčné prešlo počas práce vývojom. Vytvorené boli dve verzie riešenia: prvé opäť takzvané naivné riešenie, ktoré neoptimalizuje využívanie pamäte a opäť pracuje s plnými symetrickými maticami. Takisto aj výpočet vlastných čísel a vektorov nie je optimálny. Druhé riešenie je paralelnou verziou algoritmu popísaného v podkapitole 7.2.2, rovnako efektívne vo využívaní pamäte a minimalizujúce zbytočné výpočty.

7.3.1 Naivné riešenie

cuBLAS je knižnica základných operácií lineárnej algebry, vytvorená spoločnosťou NVIDIA. Ide o verziu štandardnej BLAS knižnice akcelerovanej cez GPU. cuBLAS je skutočne vysoko výkonná, pretože dosahuje oproti poslednej verzii MKL BLAS zrýchlenia 6 až 15 krát. Knižnica ponúka operácie s hustými maticami a vektormi, ktoré sú maximálne efektívne vo využívaní zdieľanej pamäte a zdrojov grafickej karty.

CULA je knižnica poskytujúca akcelerované implementácie najpopulárnejších operácií v lineárnej algebre. CULA implementuje funkcie z kližnice LAPACK a akceleruje ich prostredníctvom architektúry CUDA. Oproti cuBLAS má CULA implementované aj rutiny pre výpočet vlastných čísel a vektorov.

7.3.1.1 Matica Affinity Matica affinity (ďalej len matica A) sa alokuje na zariadení ako plná matica, uložené sú všetky prvky matice a výpočet prebieha aj nad duplicitnými dátami, z dôvodu že pri ďalších výpočtoch je potrebná plná veľkosť aj keď symetrickej matice.

7.3.1.2 Diagonálna matica Pri výpočte diagonálnej matice, sa využíva funkcia z knižnice cuBLAS `cublasSgemv_v2`, ktorá počíta násobenie matice s vektorom. V našom prípade produkt násobenie matice A s jednotkovým vektorom, z čoho vzniká vektor zložený z prvkov, ktoré sú súčtami riadkov matice A.

Toto riešenie je v tomto prípade úplne ideálne, pretože `cublasSgemv_v2` dosahuje pri násobení podobnú priepustnosť ako jednoduché prečítanie každej hodnoty matice, čo môže byť považované za maximálny výkon, ktorý sa dá dosiahnuť pri sumácii riadkov alebo stĺpcov matice. Operácia, ktorá je v tomto prípade navyše - prenásobenie každého prvku matice číslom 1.0 výkon veľmi neznižuje, pretože funkcia `cublasSgemv_v2`[10]:

- je v princípe viazaná priepustnosťou pamäte, jedna numerická operácia navyše nebude bottleneckom[15].
- používa inštrukcie FMA[16], ktoré zvyšujú priepustnosť inštrukcií .
- jednotkový vektor nie je pamäťovo náročný a GPU ho môže cacheovať a zvýšiť tak priepustnosť pamäte.

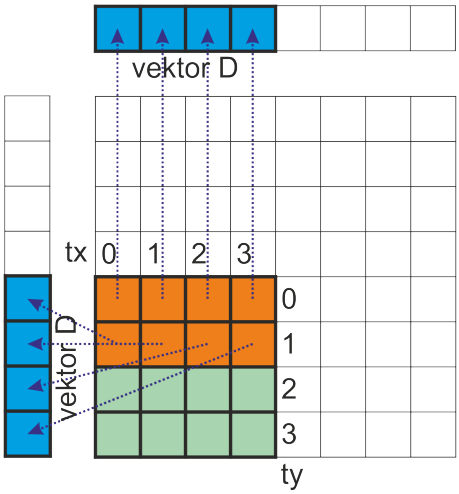
7.3.1.3 Laplaceova matica Pri vytváraní Laplaceovej matice (ďalej len matice L) je potrebný vektor D a matica A. Na výpočet prvku matice L je potrebný prvok matice A z rovnakého umiestnenia, čo značí ideálny prístup do globálnej pamäte. Ďalej sú potrebné dva prvky z vektoru D. Aby sa minimalizoval prístup do globálnej pamäte, ukladajú sa na začiatku výpočtu potrebné časti vektoru D do zdieľanej pamäte. To vykonávajú iba niektoré vlákna, konkrétne tie s `threadIdx.y` rovným 0 a 1. Tak ako je to ilustrované na nasledujúcom obrázku 12.

Kernel je konfigurovaný ako dvojrozmerný a štvorcový. Oranžovou farbou sú označené vlákna, ktoré ukladajú vektor do zdieľanej pamäte, zelenou sú označené v tej chvíli neaktívne vlákna a modrou sú označené časti vektoru D, potrebné k výpočtu fragmentu matice L.

Samotné násobenie je už potom jednoduché a optimalizované. Vlákno si vynásobí príslušné prvky z vektorov D s prvkom matice A na svojich vlastných globálnych indexoch (idx a idy) a na tie isté indexy zapíše výsledok do matice L.

[15] V informatike označovaná najslabšia časť kódu/algoritmu, ktorá spôsobuje zníženie výkonu.

[16] FMA (Fused multiply–add) je operácia, ktorá násobí a sčíta v jednom kroku a zaokrúhľuje až výsledok. Bez použitia FMA by sa vypočítalo násobenie, zaokrúhlilo sa na N bitov, potom by sa výsledok sčítal a opäť by sa zaokrúhlilo na N bitov. FMA teda redukuje počet operácií a zvyšuje presnosť eliminániou zbytočného zaokrúhľovania.



Obrázek 12: Schéma načítania hodnôt do zdieľanej pamäte.

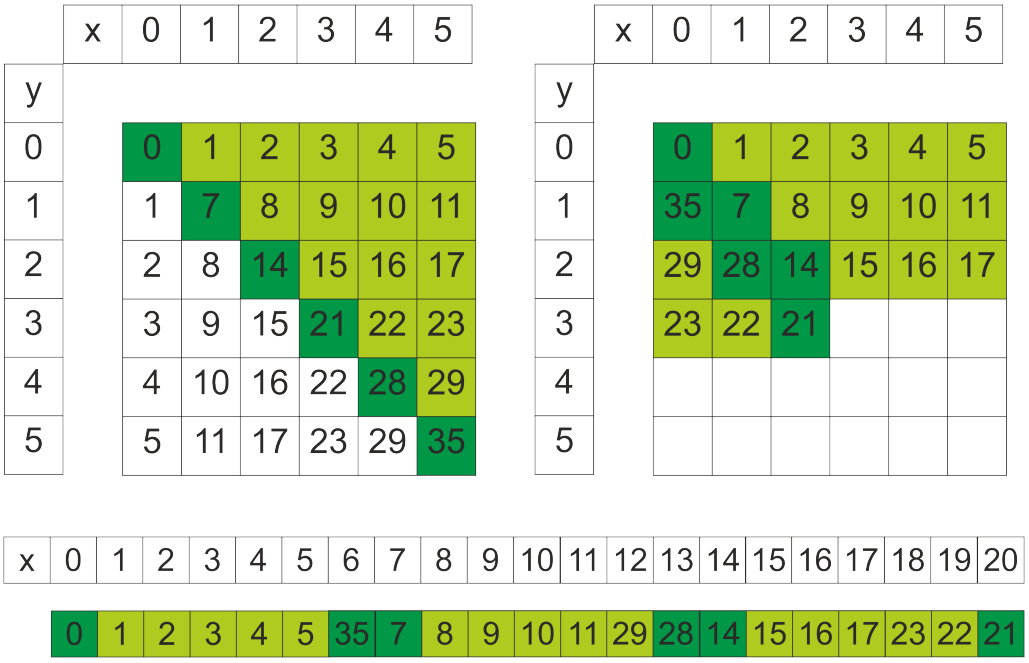
7.3.1.4 Vlastné vektory

Výpočet vlastných vektorov je riešený prostredníctvom funkcie `culaDeviceSsyevx` z knižnice CULA. Funkcia vypočíta vlastné čísla a k nim prislúchajúce vlastné vektory zo symetrickej matice. Je možné určiť počet vektorov, ktoré má CULA vypočítať. Za výpočtami stojí podobne ako v mojom paralelnom riešení metóda power iteration. Klastrovanie a premapovanie prebieha rovnako ako v sekvenčnom algoritme v kapitole 7.2 a preto nie je potrebné ich znovu rozoberať.

7.3.2 Vylepšené riešenie

Vo vylepšenom riešení išlo predovšetkým o zníženie pamäťových nárokov a o optimalizáciu výpočtov v CUDE, aby sa dosiahlo čo najvyššieho zrýchlenia oproti sekvenčnému riešeniu. To bolo v poslednej verzii veľmi dobré, preto sa vývoj paralelného riešenia odvíja od neho a v mnohých ohľadoch je mu veľmi podobné. Najväčšia zmena prichádza v spojitosti s nemožnosťou inkrementálneho indexovania vrámci kernelov, tak ako to bolo v sekvenčnom algoritme. Bolo potrebné vymyslieť systém ukladania horného trojuholníku do pamäte aby bolo možné v každom z bodov pôvodnej plnej matice dopočítať indexy v redukovanej matici. Alebo naopak z indexov v redukovanej matici vypočítať index bodu v hornom trojúholníku pôvodnej plnej matice.

Ak si predstavíme maticu, ktorá má hodnoty iba na diagonále a v hornom trojuholníku, je možné si všimnúť, že už v druhom riadku sa vytvára prázdne miesto, v ďalšom riadku už sú miesta dve atď. Pre zachovanie možnosti dohľadania indexov je potrebné sem uložiť časť matice bez zbytočnej ďalšej zmeny v jej usporiadaní. Preto bolo navrhnuté ukladanie matice ako to ilustruje obrázok 13:



Obrázek 13: Usporiadanie poľa hodnôt horného trojuholníku matice v CUDE. Prvá matica - symetrická. Druhý obrázok - dvojrozmerné usporiadanie prvkov zredukovanej matice. Spodný obrázok - usporiadanie prvkov zredukovanej matice do poľa.

Rozlišujeme dva spôsoby adresovania:

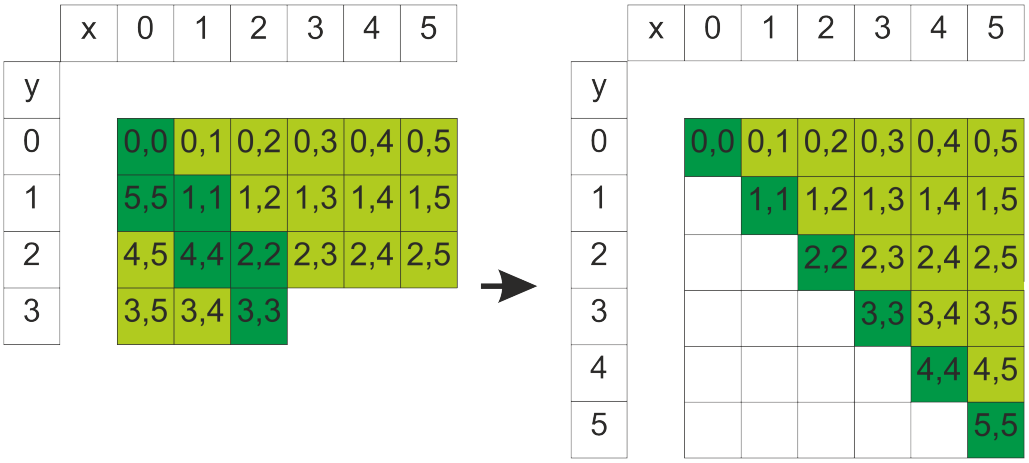
7.3.2.1 Prevod indexov minimalizovanej matice na indexy horného trojuholníku symetrickej matice Tento postup je používaný ak je potrebné vytvoriť alebo upraviť zminimalizovanú maticu. Takéto indexovanie sa využije v kernely, ktorý je spustený nad veľkosťou zminimalizovanej matice a potrebujeme vypočítať indexy vzhľadom na pôvodnú maticu, aby sme zistili napríklad polohu príslušných bodov v obrázku pri vytváraní matice Affinity.

Na obrázku 14 je znázornené premapovanie indexov x_0 a y_0 na súradnice do horného trojuholníku plnej matice. Indexy znázornené vo vnútri matíc sú už premapované a písané v tvare y_1, x_1.

V tabuľke 1 sú uvedené výpočty indexov, m je jeden z rozmerov symetrickej matice:

podmienka	x_1	y_1
$x_0 \geq y_0$	x_0	y_0
$x_0 < y_0$	$m - x_0 - 1$	$m - y_0$

Tabulka 1: Prevod súradníc.

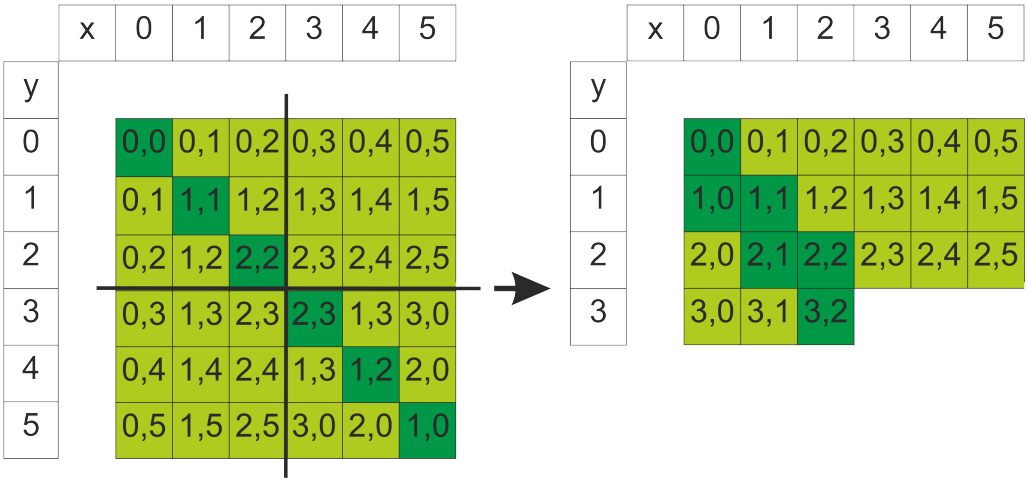


Obrázek 14: Premapovanie súradníc z minimalizovanej matice na horný trouholník symetrickej matice.

7.3.2.2 Prevod indexov symetrickej matice na indexy minimalizovanej matice

Tento postup sa využíva ak je potrebné vo výpočte zrekonštruovat'plnú symetrickú maticu, ako napríklad v prípade výpočtu diagonálnej matice. Preto musí byt'kernel spustený nad plnou veľkosťou symetrickej matice.

Na obrázku 14 je znázornené premapovanie indexov x_0 a y_0 na súradnice do zredukovanej matice. Indexy znázornené vo vnútri matíc sú už premapované a písané v tvare y_1, x_1.



Obrázek 15: Premapovanie súradníc z plnej symetrickej matice na indexy do minimalizovanej matice.

V tabuľke 2 sú uvedené výpočty indexov, m je jeden z rozmerov symetrickej matice:

podmienky			x_1	y_1
$x_0 \geq y_0$	&&	$y_0 < \lfloor (m+1)/2 \rfloor$	x_0	y_0
$x_0 \geq y_0$	&&	$y_0 \geq \lfloor (m+1)/2 \rfloor$	$m - x_0 - 1$	$m - y_0$
$x_0 < y_0$	&&	$x_0 < \lfloor (m+1)/2 \rfloor$	y_0	x_0
$x_0 < y_0$	&&	$x_0 \geq \lfloor (m+1)/2 \rfloor$	$m - y_0 - 1$	$m - x_0$

Tabulka 2: Prevod súradníc.

7.3.2.3 Matica Affinity Po zadefinovaní indexovania je možné bez vačších problémov pracovat's pamäťou vrámci kernelu. Tak ako to bude ukázané na kernely výpočtu matice Affinity (ďalej len matice A). Ten je spúšťaný iba nad veľkosťou zredukovanej matice A a preto využívame prvé indexovanie z kapitoly 7.3.2.1.

```
__global__ void kernel_get_A(TYPE_REAL * d_mat, uchar4 * d_image, int mat_cols, int mat_size )
{
    unsigned int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
    unsigned int idy = (blockIdx.y * blockDim.y) + threadIdx.y;
    unsigned int index = idy * mat_cols + idx;

    if ((idx < mat_cols) && (index < mat_size))
    {
        // remap idx, idy to x, y
        unsigned int x = (idx >= idy)? idx : mat_cols - idx - 1;
        unsigned int y = (idx >= idy)? idy : mat_cols - idy;

        uchar4 a = d_image[x];
        uchar4 b = d_image[y];

        d_mat[index] = sqrt((float)(SQR(a.x - b.x) + SQR(a.y - b.y) + SQR(a.z - b.z)));
    }
}
```

Výpis 4: Výpočet matice Affinity v kernely

Obrazové dáta sú uložené v dátovom type uchar4, kde na prvých troch miestach je uložená jedna farebná zložka a posledný byte je voľný. Tento dátový typ je výhodnejší ako uchar3 z dôvodu zarovnania dát v pamäti a možnost' využitia cacheovania a splývania pamäťových prenosov.

Pri premapovaní indexov sú využité ternárne operátory, ktoré opät' trochu znižujú počet inštrukcií oproti koncepcii `if-else`.

7.3.2.4 Diagonálna matica Pri výpočte diagonálnej matice je využívaný druhý typ popísaného indexovania z podkapitoly 7.3.2.2. Vzniká ale problém so sčítavaním prvkov, pretože vlákna nedokážu paralelne upravovat' jednu sumu. Takýto prístup by sa zaserializoval a v konečnom dôsledku by každé vlákno prepísalo predchádzajúci výsledok. Z tohto dôvodu bolo navrhnuté riešenie pomocou paralelnej redukcie. Ide o techniku kedy len niektoré vlákna sčítavajú postupne dve hodnoty až do bodu kedy vznikne jeden výsledok.

Toto riešenie pracuje so zdieľanou pamäťou aby bolo možné pristupovať k výsledkom výpočtu aj iných vlákien bloku.

```
if ((idx < mat_cols) && (idy < mat_cols))
{
    if (idx >= idy){
        x = (idy < half)? idx : mat_cols − idx − 1;
        y = (idy < half)? idy : mat_cols − idy;
    }
    else{
        x = (idx < half)? idy : mat_cols − idy − 1;
        y = (idx < half)? idx : mat_cols − idx;
    }

    s_mat[index] = d_mat[y ∗ mat_cols + x] ∗ d_mat[y ∗ mat_cols + x];

    if (tx < 16) s_mat[index] += s_mat[index + 16];
    if (tx < 8) s_mat[index] += s_mat[index + 8];
    if (tx < 4) s_mat[index] += s_mat[index + 4];
    if (tx < 2) s_mat[index] += s_mat[index + 2];
    if (tx == 0) d_buffer[blockIdx.x ∗ mat_cols + idy] = s_mat[index] + s_mat[index + 1];
}
```

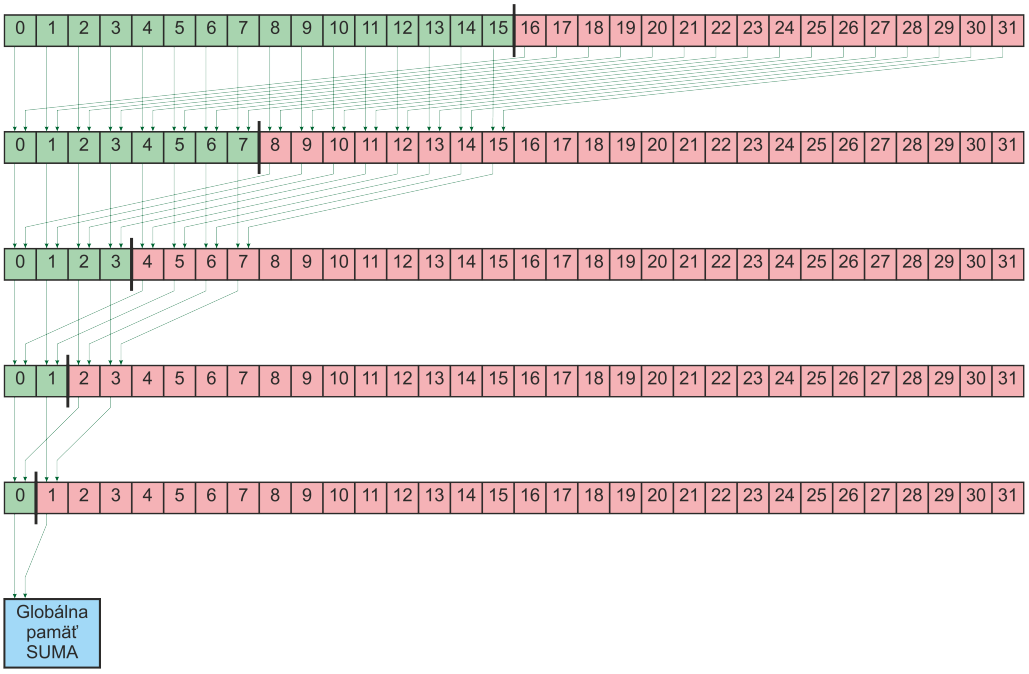
Výpis 5: Časť kernelu výpočtu Diagonálnej matice

Tento kernel je spúšťaný nad veľkosťou nezredukovanej matice A. Blok je konfigurovaný na 32 x 8 vlákien. Na začiatku sa vyalokuje zdieľaná pamäť tak, aby každé vlákno malo jedno pamäťové miesto, kam vlákno uloží umocnený prvok z vypočítaných súradníc. Vzniká tak pomyselná matica o veľkosti 32 x 8. V tomto bode sa zredukujú aktívne vlákna na polovicu a sčítajú svoju hodnotu s hodnotou jedného z vlákien, ktoré je vypnuté. Vypína sa ďalšia polovica aktívnych vlákien a proces sa opakuje až kým nezostanú v bloku aktívne iba vlákna s `threadIdx.x` rovným nule. Tieto vlákna zapíšu výsledok do pripraveného bufferu v globálnej pamäti. Do bufferu sú ale výsledky zapisované tak že medzivýsledky jedného riadku sa zapisujú pod seba do stĺpca, kvôli lepšiemu prístupu v ďalšom kroku. Grafické znázornenie redukcie je na obrázku 16.

Celý postup by mohol byť ďaleko efektívnejší s použitím spúšťania nového kernelu, ktorý by vždy sčítal dve čísla a znovu sa spustil a zabránil tak neaktívnym vláknam. Avšak takýto prístup vyžaduje ďalšiu pamäť zariadenia s veľkosťou polovice spracovávanej matice. Z tohto dôvodu je toto vylepšenie nevhodné, pretože cieľom bolo maximalizovanie veľkosti dát, ktoré dokáže riešenie spracovať.

Popisované riešenie samozrejme na medzivýsledky potrebuje ďalšiu globálnu pamäť, ale tá je zredukovaná na $\frac{1}{32}$ veľkosti spracovávanej matice. V tomto bode máme maticu v globálnej pamäti, v ktorej je potrebné sčítať stĺpce. Uloženie v stĺpcoch je dôležité, pretože vlákna budú pristupovať k dátam zarovnane a dôjde ku prekrytiu latencie globálnej pamäte. Samotné sčítanie už je v kernely riešené cyklom, ktorý prejde všetky riadky. Vždy ale vlákna pristupujú k pamäti zarovnane. Výsledkom tejto operácie je vektor D.

7.3.2.5 Laplaceova matica Laplaceova matica sa princípom výpočtu neodlišuje od výpočtu matice A, indexovanie a konfigurácia kernelu zostáva rovnaká.



Obrázek 16: Princíp sčítania prvkov v matici D použitím paralelnej redukcie.

7.3.2.6 Vlastné vektory Výpočet vlastných vektorov je bezpochyby v tomto algoritme tá najnáročnejšia časť. Preto bolo dôležité sparalelizovanie tých najnáročnejších pasáží algoritmu power iteration. Celá metóda je detilne popísaná v podkapitole 7.2.2.4. Ďalej výkonu riešenia napomáha eliminácia zbytočných prenosov z hostu na zariadenie a naopak. Na druhej strane ale neboli paralelizované časti kódu, ako sčítanie hodnôt vektoru, pretože operácia sumy poľa nie je pre CUDU úplne ideálna, a použitie paralelnej redukcie by malo zmysel pri sčítaní prvkov vektoru o veľkosti v rádoch miliónov prvkov a viac, naše vektory nemajú viac ako 20 000 prvkov. Réžia spúšťania kernelov by efektivitu znižovala na úroveň sekvenčného výpočtu. Preto boli sparalelizované dve najzásadnejšie časti metódy power iteration:

- **Deflácia Laplaceovej matice** - ide upravovanie redukovanej matice L, preto sa využíva prvé adresovanie z podkapitoly 7.3.2.1. Postup je opäť podobný ako pri vytváraní matice A. Kernel je upravený len o numerické výpočty, ktoré odčítavajú z Laplaceovej matice vlastné čísla a vektory.

- **Násobenie Laplaceovej matice a vektoru** q - Operácia násobenia matice a vektoru produkujúca ďalší vektor. V podstate ide o sčítanie prvkov v riadkoch matice, ktoré sa prenásobia príslušným prvkom vektoru. V princípe je táto operácia rovnaká ako vytvorenie matice D. Jediný rozdiel je v pridanom násobení vektorom. Tento vektor je ukladaný do zdieľanej pamäte, pretože ho vo výpočte používajú viackrát vlákna, ktoré sú v bloku umiestnené pod sebou. Kernel je konfigurovaný rovnako ako ten pri výpočte vektoru D - 32 x 8 vlákien. Preto operácia načítania vektoru z globálnej

pamäte do zdieľanej pamäte zredukuje prístup do globálnej pamäte 8x. Ďalší postup už je totožný s postupom výpočtu matice D.

Týmto postupom sa podarilo vyhnúť zbytočnému prenosu obrovskej matice L medzi zariadením a hostom. Ako už bolo spomenuté skôr, sčítavanie prvkov vektoru nebolo paralelizované. Preto pre výpočet normy a rozdielu vektoru q medzi iteráciami je potrebné tento vektor z pamäte zariadenia vykopírovať. Táto operácia má ale zanedbateľný vplyv na celkový čas.

Po výpočte vlastných vektorov, sa tieto ešte rozklastrujú prostredníctvom k-means. Táto operácia ale z celkového času algoritmu trvá zanedbateľnú časť a kvôli zložitosti algoritmu nebolo nutné ju paralelizovať.

8 Experimenty

Pre zistenie efektivity paralelného riešenia budú algoritmy porovnávané z hľadiska časovej náročnosti. Pre praktické merania som použil svoj domáci počítač osadený procesorom AMD Phenom X4 965, ktorý pracuje na základnej frekvencii 3400 Megaherzov a disponuje L2 cache 2 MB a L3 cache 6MB. V systéme je osadených 6 Gigabytov pamäte RAM. Na meranie výkonu paralelného riešenia bola použitá grafická karta založená na architektúre Fermi - GeForce GTX 560Ti s 384 CUDA jadrami, procesorom s frekvenciou 1645 Megaherzov a využíva pamäť GDDR5 s teoretickou rýchlosťou 128 Gigabajtov/sekundu a kapacitou 1 Gigabyte. Compute capability verzia 2.1. Ako vývojové prostredie bolo použité Microsoft Visual Studio 2010 Ultimate, v systéme boli nainštalované NVIDIA ovládače verzie 334.68 a CUDA verzie 6.0.1.

8.1 Meranie Času

Pre zistenie efektivity algoritmov bolo potrebné merať čas, za ktorý sú schopné vstupný obraz rozklastrovať. Na to veľmi dobre poslúžila funkcia `clock()` z knižnice `time.h`, ktorá zaznamenáva poradie aktuálneho hodinového cyklu procesoru. Na začiatku teda zaznamená aktuálty cyklus, po skončení algoritmu zaznamená opäť cyklus. Rozdiel týchto hodnôt je počet cyklov procesoru strávených prácou na algoritme. Na prepočet do časových jednotiek, v tomto prípade sekúnd, je potrebné hodnotu vydeliť počtom cyklov za sekundu, čo je hodnota frekvencie procesoru v herzoch.

Pri meraní výsledných časov sa musí počítať s malými odchýlkami, ktoré sú spôsobené nekonštantým vyťažením systému. Tieto odchýlky boli minimalizované väčším počtom meraní času, ktorý sa vo výsledku spriemeroval. Ďalší problém s presnosťou merania by mohli spôsobovať úsporné funkcie moderného hardwaru, ktoré ale boli v čase merania úplne vypnuté a procesor ako aj grafická karta išli aj v dobe nečinnosti na svojich plných taktoch. V prípade tohto algoritmu ale odchýlky nebudú nijak výrazné, pretože samotný výpočet je časovo náročný (až v rádoch desiatok sekúnd) a milisekundy navyše spôsobené systémom, nezohrávajú veľkú úlohu.

Časy boli merané v režime release a na rôznych obrázkoch v rôznych rozlíšeniach, aby boli dobre viditeľné výkonnostné rozdiely medzi sekvenčným a paralelným algoritmom a vplyv veľkosti vstupných dát ako aj ich typ. Meranie zahŕňa celý algoritmus od načítania vstupného obrazu, až po vykreslenie výsledku v závere. Relatívne zrýchlenie je možné si predstaviť ako počet iterácií, ktoré dokáže paralelný algoritmus dokončiť za jednu iteráciu sekvenčného algoritmu.

8.2 Namerané hodnoty

Z nameraných časov je vidno, že paralelná verzia je skutočne veľmi výkonná a podáva zrýchlenia až viac ako 6X, vyššie zrýchlenia sú pri spracovávaní väčšieho objemu dát.

V tabuľke sú aj hodnoty priradenej pamäte RAM, paralelná verzia opäť ukazuje, že jej hospodárenie s pamäťou je výborné - používa jej približne 4x menej ako sekvenčný algoritmus.

súbor	[px]	typ	t [s]	RAM [MB]	GPU RAM [MB]	nezred. [MB]	$\frac{CPU}{GPU}$
april.png	220x93	CPU GPU	77,554 12,995	807,467 216,715	- 848,427	1596,877	5,968
stop_sign.png	170x114	CPU GPU	96,416 15,865	725,598 171,043	- 761,207	1432,741	6,077
fei_logo.jpg	112x133	CPU GPU	60,348 10,832	431,133 87,531	- 449,732	846,446	5,571
photo.jpg	130x154	CPU GPU	91,021 18,093	773,227 181,070	- 794,393	1528,932	5,030

Tabulka 3: Výsledky.

V predposlednom stĺpci sú uvedené hodnoty, ktoré by zaberala v pamäti iba samotná matica Affinity, pri spracovávaní daného obrázku bez minimalizácie. Treba si ešte uvedomiť, že v pamäti je potrebné mať uložené aj iné matice a vektory. V porovnaní s riešením, či už paralelným alebo sekvenčným, kde sú hodnoty vyťaženia pamäte konečné, je hodnota veľkosti nezredukovanej matice obrovská.

8.3 Testovacie obrázky

(a) april

(b) Segmentácia k=2

(a) stop_sign

(b) Segmentácia k=3

(a) fei_logo

(b) Segmentácia k=3

(a) photo

(b) Segmentácia k=3

9 Diskusia

Problematika spektrálneho klastrovania je veľmi široká a pre mňa veľmi zaujímavá. Na začiatku práce som nevedel o klastrovaní takmer nič. Ale popri vývoji a práci na sekvenčnom alebo paralelnom riešení, som sa dostal hlbšie do tematiky. Pochopil som princíp metódy spektrálneho klastrovania ako aj power iteration. Teraz poznám aj teoretické pozadie používaných postupov a celá koncepcia klastrovania mi do seba výborne zapadá.

Spektrálne klastrovanie, ktoré som spracovával v tejto práci bolo testované na obraze, kde podáva skutočne výborné výsledky. Ale obrázky, ktoré dokáže algoritmus spracovať sú veľmi malé, pretože pamäťová náročnosť je minimálne $O(n^2)$. Preto výsledok nieje na prvý pohľad veľmi ohromujúci pre človeka, ktorý sa v danej tematike neorientuje. Výborné využitie ale vidím napríklad v algoritme na ktorom som pred časom pracoval, ide o program ktorý vytvára a usporiadava graf na základe podobnosti uzlov.

Na riešení, ktoré bolo predstavené v tejto práci by som chcel ďalej pracovať a úspešne ho rozšíriť o možnosť spracovania riedkych matíc. Pri vývoji riešenia som pracoval na metódach, ktoré pracujú s riedkymi maticami, avšak objavili sa problémy, kvôli ktorým som toto riešenie nedokončil. Pri deflácii riedkej matice je potrebné znižovať aj prvky matice, ktoré sú nulové, to znamená že z riedkej matice sa opäť stáva matica hustá.

10 Záver

Manipulovanie a spracovanie obrazu je jedným z mojich koníčkov a preto ma táto práca veľmi bavila. Spektrálne klastrovanie, ktoré som v práci spracovával nieje triviálnou metódou a k jej pochopeniu je potreba hlbších znalostí v oblasti lineárnej algebry. Pri práci som metódu spektrálneho klastrovania pochopil veľmi dobre, čo mi umožnilo navrhnúť sekvenčné riešenie, ktoré je veľmi výkonné a úsporné z hľadiska pamäťovej náročnosti.

Paralelné riešenie je vylepšenou paralelnou verziou už tak výborného sekvenčného riešenia. Architektúra CUDA ukázala, že aj na mainstreamovej grafickej karte, je možné dosiahnuť skvelé časy a zrýchlenia výpočtov. Pri vývoji som ale narazil na rôzne obmedzenia a ťažkosti plynúce z priority maximalizovania množstva dát, ktoré sa budú dať algoritmom spracovať. Všetky problémy sa ale podarilo vyriešiť a vo výsledku paralelné riešenie podáva výborné výkony. Ak by bola veľkosť spracovávaných dát menšia, prípadne by bola kapacita pamäte grafickej karty mnohonásobne vyššia, bolo by možné dosiahnuť lepšie výsledky, ale na úkor pamäťovej náročnosti.

11 Reference

[1] ALMASI, George S a Allan GOTTLIEB. *Highly parallel computing*. Redwood City, Calif.: Benjamin/Cummings, c1989, xxiv, 519 p. ISBN 08-053-0177-1.

[2] NVIDIA CUDA C Programming Guide. NVIDIA. *NVIDIA* [online]. 2014 [cit. 2014-04-24]. Dostupné z: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[3] CUDA Parallel Computing Platform: History of GPU Computing. NVIDIA. *NVIDIA* [online]. 2013 [cit. 2014-04-24]. Dostupné z: http://www.nvidia.com/object/cuda_home_new.html

[4] SHAPIRO LINDA, G. *Computer vision*. Vyd. 1. New Jersey: Prentice-Hall, 2001, 580 s. ISBN 01-303-0796-3.

[5] ZHENG, Jing, Weimin ZHENG, Wenguang CHEN, Yurong CHEN, Yimin ZHANG a Ying ZHAO. *Parallelization of Spectral Clustering Algorithm on Multi-core Processors and GPGPU* [online]. Hsinchu, 2008, 2008-08-06 [cit. 2014-04-24]. ISBN 978-1-4244-2683-6. Dostupné z: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4625449. Odborná. Tsinghua University.

[6] MacQueen, J.B.: *Some methods for classification and analysis of multivariate observations*. In: Cam, L.M.L., Neyman, J. (eds.) Proc. of the

fth Berkeley Symposium on Mathematical Statistics and Probability. vol. 1, pp. 281-297. University of California Press (1967)

[7] GOLUB, Gene H. *Matrix computations*. 3rd ed. Baltimore: Johns Hopkins University Press, c1996, s. 330-332. Johns Hopkins studies in the mathematical sciences. ISBN 0-8018-5414-8.

[8] DONALD, Allen. POWER METHOD. In: *NUMERICAL CALCULATION OF EIGENVALUES* [online]. Department of Mathematics Texas A&M University College Station, 2003 [cit. 2014-04-28]. Dostupné z: http://www.math.tamu.edu/~dallen/linear_algebra/chpt6.pdf

[9] ROBERTS, Stephen a Michaelmas TERM. Computation of matrix eigenvalues and eigenvectors. In: *ENGINEERING COMPUTATION* [online]. 2005 [cit. 2014-04-29]. Dostupné z: http://www.robots.ox.ac.uk/~sjrob/Teaching/EngComp/ecl4.pdf

[10] *Applied parallel and scientific computing: 10th International Conference, PARA 2010, Reykjavik, Iceland, June 6-9, 2010, revised selected papers, part I*. 1st ed. New York: Springer, 2012, p. cm. ISBN 36-422-8150-8. Dostupné z: http://books.google.cz/books?id=ip_RTivR_5gC&printsec=frontcover&hl=sk&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false

A Príloha na CD/DVD

Teoretická časť:

- Práca.pdf

Praktické riešenie:

- 1.Sekvenčné riešenie - naivné
- 2.Sekvenčné riešenie - vylepšené
- 1.Paralelné riešenie - naivné
- 2.Paralelné riešenie - vylepšené